Fakultät EIM, Institut für Informatik
AG Mensch-Computer-Interaktion und Softwaretechnologie
Universität Paderborn

STUDIENARBEIT

GENERIERUNG VON KORREKTEN UND MINIMALEN OBERFLÄCHENMODELLEN AUS MASCHINENMODELLEN

Henning Wachsmuth
henningw@upb.de

vorgelegt bei
Prof. Dr. Gerd Szwillus und Prof. Dr. Franz Josef Rammig

ABSTRACT

Bislang beruht der Prozess des User Interface Designs auf intuitivem Vorgehen und dem Rückgriff auf Heuristiken. Dieses Vorgehen birgt jedoch Schwachstellen, denn es zeigt sich immer wieder, dass bei der Interaktion zwischen Mensch und Maschine Situationen auftreten, in denen das Interface falsche Aussagen über die internen Abläufe des unterliegenden Systems macht, was im Ernstfall schwerwiegende Folgen haben kann.

Die Wissenschaftler Michael Heymann und Asaf Degani haben daher ein Verfahren entworfen, das zu dem gegebenen Modell eines Systems und der Aufgabenspezifikation auf formalem Weg ein Modell der Oberfläche generiert, aus dem die anzuzeigenden Informationen eines Interfaces direkt ableitbar sind. Ziel ist, die Beobachtbarkeit des unterliegenden Systems zu gewährleisten; der Benutzer muss jederzeit wissen, in welchem Zustand sich das System befindet und welchen es in Abhängigkeit zur nächsten Aktion erreichen wird. Außerdem sollen nur die dafür minimal notwendigen Informationen vermittelt werden.

Im Rahmen dieser Arbeit geht es darum, das Verfahren abstrakt wie auch anschaulich vorzustellen und in einem graphischen Software-Tool umzusetzen. Darüber hinaus deckt die Arbeit Fehler im Verfahren auf, welche zu User Interfaces führen, die nicht den Forderungen von Heymann und Degani gerecht werden. Anschließend werden Verbesserungen aufgezeigt, aus denen die Korrektheit des Verfahrens folgt.

INHALTSVERZEICHNIS

1 EINLEITUNG

„Knowing what to abstract and how to abstract it is the foundation of any user interface design.“

(DEGANI 2004, S. 31)

1.1 MOTIVATION – EINFÜHRUNG IN DAS THEMA

Kaum ein Bereich des öffentlichen wie auch privaten Lebens ist in heutigen Tagen ohne die Einbindung technischer Geräte vorstellbar. Im eigenen Wohnzimmer, in Geschäften und Banken, in öffentlichen Gebäuden und insbesondere auch am Arbeitsplatz ist der Mensch auf die Interaktion mit Maschinen und Programmen angewiesen. Einfache Heim- und Alltagsgeräte wie Fernseher, Mikrowelle, Geld- oder Fahrkartenautomat, ebenso wie komplexe Systeme, die sich beispielsweise in Handys, Fahrzeugen und Personal Computern finden lassen, haben dabei eins gemeinsam: der Benutzer interagiert mit der Maschine über eine Schnittstelle.
Solch ein *User Interface* – sei es die virtuelle Oberfläche eines Programms oder die Knöpfe und Anzeigen eines Bedienpults – lässt sich als Abstraktion des unterliegenden Systems begreifen. Diese Abstraktion ist notwendig, denn oftmals beinhalten die systeminternen Abläufe hunderte, wenn nicht gar tausende von Zuständen, die durch zahlreiche manuell gesteuerte oder automatisch auftretende interne und externe Ereignisse beeinflusst werden. Für die Erfüllung der Aufgaben ist in der Regel jedoch nur ein sehr geringer Ausschnitt der enthaltenen Informationen relevant.
Eine Oberfläche zeigt dem Benutzer also bestimmte Daten, verbirgt andere vor ihm. Die Ausgabe dient dem Benutzer als Information, die er zur Steuerung des Systems benötigt, und auch die Möglichkeiten des Einflusses auf das Verhalten hängen unmittelbar von den Gegebenheiten des User Interfaces ab.

Es stellt sich die Frage, welche Informationen nach außen getragen werden müssen, damit die *korrekte* Bedienbarkeit einer Maschine gewährleistet ist. Der Benutzer sollte jederzeit in der Lage sein zu erkennen, in welchem Zustand sich das System befindet und was etwaige Handlungen für Konsequenzen haben. Das mag bei einem simplen Radiowecker noch relativ belanglos erscheinen, bei sicherheitskritischen Systemen wie dem Autopiloten eines Flugzeugs entscheiden diese Kriterien aber mitunter über Leben und Tod. In dem Buch „Taming HAL: Designing Interfaces Beyond 2001“ (DEGANI 2004) beschreibt Asaf Degani vom *NASA Ames Research Center* in Kalifornien mehrere solcher Fälle.
Ausreichende Information ist jedoch nur die eine Seite. Genauso sollte der Benutzer nicht mit irrelevanten Informationen konfrontiert werden, denn sie stellen eine unnötige kognitive Belastung dar. „There is a clear advantage for simpler interfaces – they usually are cheaper to

produce, they make user manuals smaller and less confusing, and place less of a burden on the user" (DEGANI 2004, S. 161).
Eine Schnittstelle genügt im optimalen Fall also völlig unabhängig aller ästhetischen und ergonomischen Gesichtspunkte den Bedingungen, dass einerseits die erfolgreiche Ausführung aller spezifizierten Aufgaben bestmöglich unterstützt wird und andererseits der Umfang der angezeigten Zustände und Ereignisse auf ein *Minimum* reduziert ist. Daher erweist sich die sinnvolle Abstraktion, also der Prozess des Analysierens, welche Daten wesentlich und welche unwesentlich sind, als ein fundamentaler Aspekt des User Interface Designs, der letztlich zur Festlegung der im Interface anzuzeigenden Informationen führt (vgl. HEYMANN & DEGANI 2006, S. 2).

1.2 ZIELE DIESER ARBEIT

Die gegenwärtige Situation besteht darin, dass die Erzeugung eines User Interfaces keinen formal definierten Methoden folgt, sondern auf intuitivem Vorgehen beruht, was die Gewährleistung der Benutzbarkeit äußerst schwierig gestaltet. „Currently, this abstraction is performed in a heuristic and intuition-based manner, and its evaluation process usually involves many interface design iterations, costly simulations, and extensive testing" (HEYMANN & DEGANI 2006, S. 3).
Michael Heymann vom *Technion, Israel Institute of Technology* und Asaf Degani haben nun jüngst zwei Verfahren zur Verifikation und zur Generierung von User Interfaces entwickelt. Während das eine dazu dient, konkrete Schnittstellen auf ihre korrekte Abstraktion bezüglich des zugehörigen Systems hin zu untersuchen, soll das andere ermöglichen, zu einer gegebenen Maschine und der Aufgabenanforderung automatisch eine korrekte und minimale Schnittstelle zu erzeugen. Dabei werden Maschinen und Programme mittels Zustandsübergangsdiagrammen modelliert und durchlaufen danach mathematische Algorithmen.

Ziel der vorliegenden Arbeit ist auf der einen Seite, diese Verfahren vorzustellen, sie formal zu beschreiben und anschaulich zu erläutern. Vor allem die Generierung von Interfaces steht hierbei im Fokus. Dieser Teil basiert auf den Papern „Formal Analysis and Automatic Generation of User Interfaces: Approach, Methodology, and an Algorithm" (HEYMANN & DEGANI 2006) sowie „On Abstractions and Simplifications in the Design of Human-Automation Interfaces" (HEYMANN & DEGANI 2002 I). Er wird darüber hinaus aber um weitere Quellen sowie eigene Erarbeitungen erweitert werden, die sich insbesondere mit der vermeintlichen Korrektheit und der Anwendbarkeit des Generierungsverfahrens beschäftigen.
Auf der anderen Seite ist im Rahmen dieser Arbeit ein Software-Tool zu entwickeln, mit dem sich das Modell einer Maschine in einem graphischen Editor erstellen lässt, um daraus anschließend mittels des angesprochenen Verfahrens ein korrektes und minimales Modell der zugehörigen Bedienungsoberfläche automatisch generieren zu können. Die Umsetzung der Algorithmen und technische Hintergründe sind ebenfalls Inhalt des schriftlichen Teils und stellen somit die Verbindung zwischen Text und Programm dar.

1.3 GLIEDERUNG

Kapitel 2 behandelt mögliche Fehlerquellen bei der Interaktion zwischen Mensch und Maschine, die im vorliegenden Kontext relevant sind. Es wird herausgearbeitet, dass die Beobachtbarkeit eines Systems von zentraler Bedeutung für dessen Benutzbarkeit ist. Darauf aufbauend werden Ziele für diesen Bereich des User Interface Designs definiert.
In Kapitel 3 wird die Modellierung von Maschinen und Oberflächen durch Zustände und Ereignisse sowie der Aufgabenspezifikation durch so genannte Spezifikationsklassen vorgestellt. Ausgehend von intuitiven Analysetechniken findet sich dann eine ausführliche Beschreibung des Verfahrens zur Verifikation eines User Interfaces, das am Beispiel eines halbautomatischen Heiz- und Kühlsystems graphisch und inhaltlich veranschaulicht wird.
Kapitel 4 behandelt den Kernaspekt dieser Arbeit. Die Generierung des Modells der Oberfläche wird als Reduktion des Modells der Maschine herausgestellt, außerdem werden mathematische Prinzipien, auf die sich der Reduktionsalgorithmus stützt, erläutert. Anhand des Heiz- und Kühlsystems wird die Entwicklung des Oberflächenmodells Schritt für Schritt durchlaufen. Danach wird aufgedeckt, dass das Verfahren in der gegebenen Form nicht korrekt ist, und aufgezeigt, welche Verbesserungen möglich sind. Abschließend werden die Grenzen des Vorgehens analysiert.
Kapitel 5 widmet sich der technischen Umsetzung des Software-Tools. Neben der Architektur und den Funktionalitäten des Systems beschreibt dieser Abschnitt, wie sich mit dem Programm der Weg von der Erstellung eines Maschinenmodells bis hin zur Nachbearbeitung des Modells der Benutzeroberfläche umsetzen lässt.
Eine Zusammenfassung der erarbeiteten Ergebnisse ist schließlich Inhalt von Kapitel 6, ebenso wie ein Ausblick auf weitere Forschungsansätze und etwaige Erweiterungen der Software.

2 GRUNDLEGENDE PROBLEME UND ZIELE DES USER INTERFACE DESIGNS

„Humans do make mistakes, but with proper design,
the incidence of error and its effects can be minimized.“

(NORMAN 1989, S. IX)

2.1 MENSCHLICHES HANDELN UND DARAUS RESULTIERENDE FEHLERQUELLEN

Der Umgang mit dem Computer und anderen technischen Geräten ist für viele Menschen zur Selbstverständlichkeit geworden. Meist verläuft die Bedienung einer Maschine problemlos, vor allem wenn es sich um gewohnte Vorgänge handelt. Dennoch ergeben sich immer wieder Konfliktsituationen, in denen sich ein Gerät oder Programm anders als erwartet verhält oder in denen es zu Fehlverhalten seitens des Benutzers kommt. Sie gründen sich in einer großen Bandbreite an Ursachen, die von äußeren Einflüssen wie etwa klimatischen Verhältnissen über die generellen Grenzen menschlicher Leistungsfähigkeit bis hin zu eindeutigen Programmier- oder Konstruktionsfehlern des Systems bzw. der Maschine reichen.
Es wäre vermessen, alle von ihnen aufzählen geschweige denn genauer untersuchen zu wollen. Eingeschränkt auf den direkten Kontext der Mensch-Maschine-Interaktion aber lassen sich zwei grundsätzliche Kategorien von Problemtypen unterscheiden: diejenigen, die als Quelle menschliches Versagen in irgendeiner Form aufweisen, und diejenigen, die aus Design-Fehlern des User Interfaces resultieren. Insbesondere zweitere sind in dieser Arbeit von Interesse, doch stehen sie nicht selten in Verbindung mit den Fähigkeiten eines Benutzers und so wird im Folgenden auch ein Ausschnitt aus dem Bereich der *human factors* angesprochen. Auf die Behandlung ästhetischer Entwurfskriterien wird hingegen ausdrücklich verzichtet, da hier die Vermittlung von Information im Mittelpunkt steht. Denn sofern nicht primär die Optik für das Design ausschlaggebend ist, ist Information die Grundlage jedes guten Interfaces, nicht Ästhetik; „[...] if a product can't be used easily and safely, how valuable is its attractiveness“ (NORMAN 1989, S. VIII). Ebenso werden auch ergonomische Gesichtspunkte und solche, die das Bewegungssystem der menschlichen Informationsverarbeitung betreffen, nicht Thema dieses Textes sein.

„Users interact with systems or tools to achieve certain operational tasks“ (HEYMANN & DEGANI 2002 I, S. 2). Im Allgemeinen liegt es nicht in der Absicht eines Benutzers, sich mit einem technischen Gerät länger als nötig auseinanderzusetzen, er arbeitet vielmehr ergebnisorientiert. Donald A. Norman, emeritierter Leiter des Instituts für Kognitionswissenschaften der *University*

of California, spricht in diesem Zusammenhang in seinem Buch „The Design of Everyday Things" (NORMAN 1989) von den *seven stages of action* (VGL. NORMAN 1989, S. 45F.): Um eine Aktion auszuführen, formt der Mensch zuallererst ein Ziel (1). Als Konsequenz entwickelt er die Absicht zu handeln (2), spezifiziert für sich die Aktion (3) und führt sie aus (4). Anschließend nimmt er den resultierenden Zustand des Handlungskontextes wahr (5), interpretiert ihn (6) und evaluiert das Ergebnis (7). Diese stetig auftretende Handlungsfolge entspringt dem wie folgt definierten Aktionszyklus:

Definition 2.1.1: Der Aktionszyklus / The Action Cycle
(VGL. NORMAN 1989, S. 47)

> Der **Aktionszyklus** menschlichen Handelns besteht aus zwei Aspekten, der Ausführung und der Evaluation. Ausführung meint das aktive Umsetzen einer Handlung. Evaluation bezeichnet den Vergleich zwischen dem Ziel, das durch eine Handlung ausgelöst werden soll, und dem Resultat der Handlung.

Norman weist darauf hin, dass dieses zyklische Vorgehen – dargestellt in Abbildung 2.1.1 – keine vollständige psychologische Theorie widerspiegelt, sondern dass es eher als angenähertes Modell anzusehen ist. Es wird dennoch dafür hilfreich sein, die grundlegenden Ziele des User Interface Designs in Hinsicht auf die vermittelte Information herauszuarbeiten.

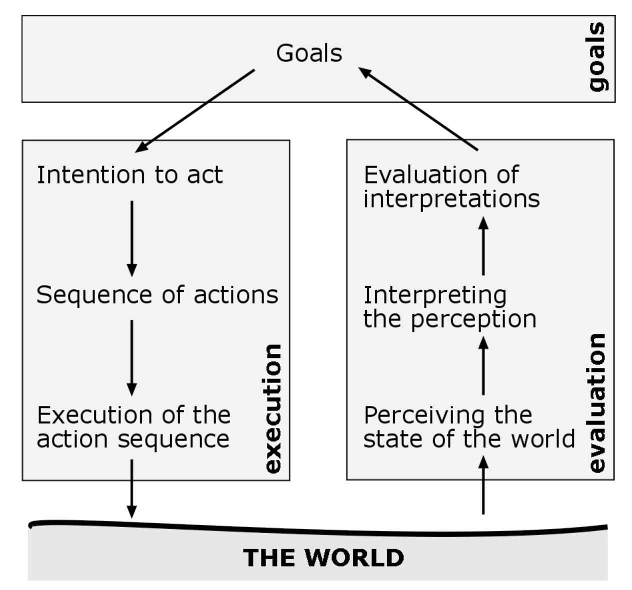


Abbildung 2.1.1: seven stages of action (NACH NORMAN 1989, S. 47)

Im Folgenden sei daher beschrieben, welche Arten von Fehlern existieren, die ein Mensch bei der Bedienung eines Systems produziert. Abgesehen wird dabei von der Beschäftigung mit der Formulierung von Handlungszielen, da dieser Bereich eng an die konkrete Aufgabenanforderung gebunden ist und somit nicht in einer generellen Analyse enthalten sein kann. Da ein falsches Ziel aber auf eine Fehlinterpretation der zuvor aufgenommenen Information hindeutet, hängt dieses Stadium unterschwellig natürlich von der Evaluation des aktuellen Zustands ab.
Während eben dieses perzeptuellen und kognitiven Prozesses können Verständnis-, Gedächtnis- und Aufmerksamkeitsfehler auftreten. Ist dem Benutzer nicht bewusst, in welchem Zustand sich das Gerät oder Programm befindet oder was die zuletzt ausgeführte Aktion für Auswirkungen

hatte, so wird er sicherlich nicht in der Lage sehen, das richtige Ziel zu formulieren. Wenn also nicht sichtbar gemacht wird, welche Daten aktuell gültig sind und der Benutzer nicht ausreichend Feedback auf Eingaben erhält, wie soll er zu den richtigen Schlüssen gelangen? Wie soll er verstehen, was das System tut? Andersherum kann die angezeigte Menge an Informationen jedoch auch unverhältnismäßig hoch sein, führt damit zu einer zusätzlichen Belastung des Gedächtnisses und Geistes. Ab einem bestimmten Grad wird der Benutzer die Schnittstelle nicht mehr überschauen können und daher nicht mehr entscheiden können, welche Informationen für ihn wesentlich sind.

Weiter steht gerade bezüglich technischer Geräte die Aufmerksamkeit des Benutzers im Fokus. Wie bereits angesprochen, stellt ein User Interface im Allgemeinen eine hohe Abstraktion der internen Logik des unterliegenden Systems dar. Damit Systeme übersichtlich bleiben, laufen zahlreiche Aktivitäten automatisch im Hintergrund ab. Insbesondere mit solchen automatisierten Systemen befasst sich diese Arbeit. „No, the transition happens without any direct user involvement; [...] this is where it all begins“ (DEGANI 2004, S. 59). Wenn Vorgänge ohne Einfluss des Benutzers geschehen, weil sie von internen oder externen Bedingungen ausgelöst werden, und sie Relevanz für die Bedienung des Systems haben, ist angemessene Rückmeldung zwingend erforderlich. In dem Paper „A Model for Types and Levels of Human Interaction with Automation“ (PARASURAMAN ET AL. 2000) betonen Raja Parasuraman, Thomas B. Sheridan und Christopher D. Wickens unter Anderem, dass der Benutzer einer Maschine bevorzugt dann aufnahmefähig ist, wenn er sie aktuell selbst steuert. Im Umkehrschluss: „Humans tend to be less aware of changes in environmental or system states when those changes are under the control of another agent (whether that agent is automation or another human) than when they make changes themselves“ (PARASURAMAN ET AL. 2000, S. 6). Auch können automatische Vorgänge zu Verwirrung führen, sogar den Anschein erwecken, das System agiere eigenständig und obliege nicht mehr den Anweisungen des Benutzers. Solcherlei Gegebenheiten bergen mögliche Fehlerquellen bei der Auswertung der gegenwärtigen Situation.

Hat der Benutzer den Zustand des Interfaces erkannt und interpretiert und darauf aufbauend sein nächstes Ziel entwickelt, ist der nächste Schritt, dieses Ziel umzusetzen. Bei der Bedienung einer Schnittstelle bedeutet das, in einer bestimmten Form eine Eingabe zu tätigen. Was aber, wenn nicht ersichtlich ist, worauf etwaige Handlungen hinauslaufen? In diesem Fall wird der Benutzer nicht erkennen können, was er tun muss, um das gewünschte Resultat zu erreichen. Einerseits ergeben sich solche Probleme dort, wo weder angezeigt wird, was der Folgezustand sein wird, noch die Bedienungsanleitung Aussagen darüber macht (sofern überhaupt auf sie zugreifbar ist), noch die intuitive Vermutung bestätigt wird. Andererseits finden sich aber auch nicht selten Interfaces, die vom Blickpunkt des Betrachters aus *nicht-deterministisch* erscheinen. Während Nicht-Determinismus einer Maschine ein im Grunde nur theoretisches Modell der Informatik ist, existiert er sehr wohl bei der Interaktion des Menschen mit einem System:

Definition 2.1.2: Nicht-Determinismus / non-determinism

(VGL. DEGANI 2004, S. 35F)

> Ein User Interface heißt genau dann **nicht-deterministisch**, wenn ein und dasselbe Ereignis β, ausgelöst in einem Zustand S_1, aus Sicht des Benutzers zwei oder mehr verschiedene Folgezustände $S_2, ..., S_k$ mit $k \geq 3$ haben kann.

Anders ausgedrückt: eine Bedienungsoberfläche ist als nicht-deterministisch zu deklarieren, wenn der Benutzer nicht vorherzusehen vermag, was in Abhängigkeit zu seiner nächsten Aktion passieren wird. Abbildung 2.1.2 verbildlicht die Definition noch einmal. „It is important to understand it because this is the fundamental problem that plagues interfaces and causes much confusion to the users" (DEGANI 2004, S. 30).

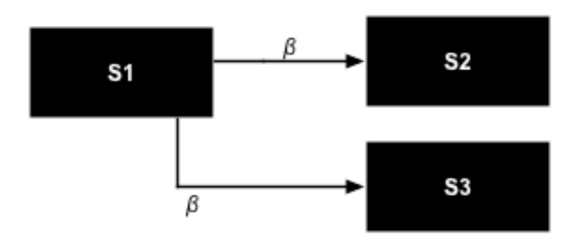


Abbildung 2.1.2: Nicht-Determinismus schematisch

Als Beispiel sei eine übliche Funktionstaste eines Mobiltelefons angeführt, die bei einem laufenden Gespräch den Abbruch von selbigem bewirkt, ansonsten aber der Einwahl ins Internet dient. Angenommen, Person *A* will ein Gespräch mit Person *B* beenden, so betätigt *A* die Taste. Hat sein Gegenüber *B*, was *A* nicht wissen kann, nun bereits kurz zuvor die gleiche Aktion ausgeführt (es kann sich hier um Bruchteile von Sekunden handeln), wird *A* fälschlicherweise das Internet betreten, ohne dass ihm das angezeigt wurde. Damit werden unnötige Kosten verursacht. Weitere Beispiele werden im Laufe dieser Arbeit folgen.

Vor allem dort, wo das Verhalten des Systems aus der Erfüllung von Bedingungen resultiert, kann ein nicht-deterministisches User Interface Verwirrung hervorrufen, im Zweifelsfall gar schlimmere Folgen haben. In diesem Fall nämlich wird sich die Erwartung bezüglich des Folgezustands manchmal bestätigen, manchmal nicht. Nicht-Determinismus tritt an vielen Stellen auf und ist oft begründet in unzureichender Anzeige oder schlechter Abstraktion. Doch er liegt nicht zwangsläufig aufgrund von Fehlern bei der Entwicklung vor, im Gegenteil: wenn es der Vereinfachung der Bedienung dient und keinen negativen Einfluss auf die Erfüllung der Aufgabe ausübt, wird eine Oberfläche oftmals bewusst nicht-deterministisch konstruiert. So zeigt etwa ein Geldautomat im Voraus nicht, in welcher Konstellation von Scheinen das angeforderte Guthaben ausbezahlt werden wird. Da dies dem Zweck nicht entgegensteht, lässt sich wenig dagegen einwenden. Doch sobald Nicht-Determinismus die korrekte Bedienung erschwert, stellt er eine unnötige Fehlerquelle dar und muss vermieden werden.

An dieser Stelle sind die wesentlichen Problemtypen, die als Grundlagen für die noch zu formulierenden Ziele dienen, ausreichend abgehandelt. Es kann daher nun definiert werden, welche Anforderungen an eine Benutzungsschnittstelle in diesem Kontext zu stellen sind.

2.2 ZIELE HINSICHTLICH DER VERMITTELTEN INFORMATION – BEOBACHTBARKEIT ALS ZENTRALE QUALITÄT

Zusammengefasst führen die bisherigen Ergebnisse dahin, dass ein User Interface so entwickelt werden sollte, dass das unterliegende Verhalten des Systems beobachtbar ist. Beobachtbarkeit

meint intuitiv gesprochen, dem Benutzer jederzeit die für die Benutzung relevanten Abläufe offen darzulegen und zwar in einer Weise, die es ihm erlaubt, das Geschehen so zu verfolgen, dass er sich in der Lage sieht, seine Ziele zu erreichen. Dies beinhaltet neben dem Wissen über den aktuellen Zustand und mögliche Folgezustände auch, dass die kognitiven Anforderungen den Fähigkeiten des Benutzers entsprechen. Der Begriff „Beobachtbarkeit" entstammt der Regelungstechnik und bezeichnet das Maß dafür, wie gut der interne Zustand eines Systems aus dem Wissen externer Ausgaben abgeleitet werden kann.
Norman schlägt die von ihm konstatierten Prinzipien guten Designs (VGL. NORMAN 1989, S. 17F. UND S. 52F) für den Entwurf eines Geräts als Leitfaden vor. Sie spiegeln implizit die grundlegenden Voraussetzungen dafür wider, dass ein Gerät beobachtbar ist: Der Benutzer kann den Zustand des Geräts und Alternativen für die nächste Aktion erkennen *(visibility)*, ebenso das Verhältnis zwischen Aktionen und Konsequenzen sowie zwischen Zuständen des Systems und des Interfaces *(good mappings)*. Außerdem erhält er stetig Rückmeldung über die Auswirkungen seiner Aktionen *(feedback)*. Weiterhin fordert Norman ein gutes konzeptuelles Modell *(good conceptual model)*, das sich aber stärker auf die optische Präsentation bezieht und daher in diesem Zusammenhang vernachlässigt wird. „Here the emphasis is on questions regarding ‚what' information must be provided to the user and ‚when' rather than on ‚how' this information is provided" (HEYMANN & DEGANI 2002 I, S. 5). Über die Beobachtbarkeit hinaus muss natürlich zusätzlich gegeben sein, dass der Benutzer zu jedem Zeitpunkt Kontrolle über das System hat, und sich insbesondere die Aufgaben, für die das System konzipiert wurde, stets erfüllen lassen.

Wie bereits erwähnt, ist die primäre Forderung, der ein User Interface im vorliegenden Kontext zu genügen hat, *Korrektheit*. Korrektheit meint, dass die Schnittstelle gewährleistet, dass der Benutzer das System korrekt bedienen kann, um seine Absichten umzusetzen. Diese Eigenschaft soll nun formalisiert werden, denn in Kapitel 3 wird es genau darum gehen, die Korrektheit einer Schnittstelle in Bezug auf die zugehörige Maschine zu verifizieren, und ebenso verkörpert sie eins der beiden Ziele der automatischen Generierung (Kapitel 4).
Unter der einzigen Annahme, dass das Verhalten der Maschine deterministisch ist und die Aufgaben des Benutzers mit ihr erfüllbar sind (VGL. HEYMANN & DEGANI 2006, S. 9), leitet sich Korrektheit direkt aus der Qualität der Beobachtbarkeit ab: Eine Benutzungsschnittstelle ist korrekt, wenn der aktuelle Zustand des unterliegenden Systems bekannt ist und sich der nächste Zustand in Abhängigkeit zur nächsten Aktion voraussagen lässt. An dieser Stelle ist wichtig zu sagen, dass ein Interface selbst diese Eigenschaften gegebenenfalls nur partiell abdecken muss, sofern die Zuhilfenahme von weiteren Materialien wie etwa einer Bedienungsanleitung zur vollständigen Erfüllung der Kriterien führt. Es hängt vom Einsatzgebiet und vom Umfang des Systems ab, inwiefern der unweigerliche Rückgriff auf externe Informationsquellen gerechtfertigt sein kann; während (einfache) Alltagsgeräte im Allgemeinen selbstbeschreibend sein sollten, erscheinen Zusatzmaterialien bei sicherheitskritischen Systemen verständliche Notwendigkeit zu sein. Nachdem im nächsten Kapitel die Repräsentation von User Interfaces durch so genannte Oberflächenmodelle eingeführt worden ist, können Anleitung und Ähnliches vernachlässigt werden, da sie unterschwellig in den Modellen integriert sein werden.
Die soeben angegebene eher inhaltliche Beschreibung des Begriffs „Korrektheit" ist für die folgenden Themen allein unzureichend. Um die noch vorzustellenden Verfahren von Asaf

Degani und Michael Heymann anwenden zu können, bedarf es nämlich einer Definition, die sich in mathematischen Modellen ausdrücken lässt und daher technischer Natur sein muss:

Definition 2.2.1: Korrektheit eines Interfaces / interface correctness
(VGL. HEYMANN & DEGANI 2006, S. 9)

Ein User Interface inklusive aller zugehörigen Hilfsmaterialien ist **korrekt**, wenn die spezifizierte Aufgabe stets erfüllbar ist und keiner der folgenden Zustände existiert:

- **error state:** Der vom Interface repräsentierte Zustand entspricht nicht dem internen Zustand des Systems.
- **restricting state:** Der Benutzer kann einen Zustandswechsel auslösen, der vom Interface nicht angezeigt und von den Hilfsmaterialien verschwiegen wird.
- **augmenting state:** Das Interface oder eins der Hilfsmaterialien verheißt einen möglichen Zustandswechsel, der intern nicht auslösbar ist.

In HEYMANN & DEGANI 2002 I wird nur von *error states* und *blocking states* gesprochen, letztere entsprechen den *restricting states* (VGL. HEYMANN & DEGANI 2002 I, S. 11). Doch nur zusammen mit den *augmenting states* erhält man die gewünschte Spezifikation. Wieso entspricht diese Spezifikation eines korrekten Interfaces der oben stehenden sprachlichen Definition? Um diese Frage zu beantworten, seien die drei gerade genannten Zustandsarten kurz genauer erklärt:
Ein *error state* ist ein Indikator für einen Widerspruch zwischen Interface und unterliegendem System bezüglich der Aufgabenanforderung. Die Bedienungsoberfläche gibt ihren Zustand nicht wahrheitsgemäß an, der eigentlich aktuelle Zustand ist dem Benutzer folgerichtig nicht bekannt; oft aber nicht zwangsläufig treten *error states* nach scheinbar nicht-deterministischen Reaktionen der Maschine auf: „[...] an error state is a direct result of non-deterministic behavior" (DEGANI 2004, S. 252).
Ein *restricting state* liegt exemplarisch vor, wenn vom ihm aus die Aktivierung des Ruhezustands eines Computers mit dessen Absturz einhergeht, ohne dass auf dieses Problem hingewiesen wird. Interfaces, die solcherlei Situationen hervorrufen, verwirren bzw. erstaunen einen Benutzer, da er mit dem Resultat nicht rechnen kann (VGL. HEYMANN & DEGANI 2006, S. 9). *Augmenting states* hingegen lassen den Benutzer an den eigenen Fähigkeiten respektive den Funktionalitäten des Systems zweifeln, denn es herrscht Unverständnis gegenüber dem Ausbleiben des gewünschten Resultats, etwa wenn das vermeintliche Auslösen einer halbautomatischen Kamera aufgrund zu geringen Lichteinfalls ohne jegliche Erklärung wirkungslos bleibt. Zusammen bilden die beiden letztgenannten die Menge an Zuständen, die falsche Aussagen über Nachfolgezustände machen.
Das Nichtvorhandensein von *error states, restricting states* und *augmenting states* bedeutet umgekehrt, dass dem Benutzer der aktuelle Zustand und die Konsequenzen möglicher Handlungen ersichtlich sind.
Einzig nicht durch die Korrektheitskriterien abgedeckt ist der Fall, dass ein internes Ereignis einen Wechsel des Zustands des User Interfaces auslöst, der dem Benutzer nicht als möglich angezeigt wird. Kapitel 4 wird sich noch damit beschäftigen, inwiefern diese Tatsache Einfluss auf die korrekte Bedienbarkeit nehmen kann.

Die Einhaltung der Kriterien ist unweigerlich mit dem Ziel verknüpft, ein sicher bedienbares User Interface zu konstruieren. Doch sie allein macht ein Programm bzw. eine Maschine nicht

benutzbar. Anderenfalls wäre zum Beispiel auch ein Interface angemessen, das die kompletten systeminternen Abläufe anzeigt, was meist sowohl technisch nicht realisierbar als auch vom Belastungsaspekt aus gesehen unvertretbar ist.
Das zweite Ziel, das beim Interface Design und somit insbesondere in Kapitel 4 im Blickpunkt steht, ist folglich, die Menge angezeigter Informationen so gering wie möglich zu halten:

Definition 2.2.2: Korrektes und minimales User Interface / correct and succinct user interface
(VGL. HEYMANN & DEGANI 2002 I, S. 16)

> Ein korrektes User Interface inklusive aller zugehörigen Hilfsmaterialien ist **minimal**, wenn die Anzeige keines Zustands und ebenso keiner weiteren Information entfernt werden kann, ohne die Korrektheitseigenschaft zu verletzen.

Es sei bereits jetzt darauf hingewiesen, dass es von der Beschaffenheit des Problems abhängen kann, ob etwa Ereignisse, die nur intern von Bedeutung sind, unter Umständen auch für den Benutzer sichtbar gemacht werden sollten. Es wird sich außerdem zeigen, dass im Regelfall nicht unbedingt nur eins, sondern mehrere korrekte und minimale User Interfaces zu einem gegebenen System und der Aufgabenspezifikation existieren können. Auf diese Aussagen wird bei der Beschreibung des Verfahrens zur automatischen Generierung solcher Interfaces noch genauer eingegangen.
Nachdem die notwendigen Fachbegriffe determiniert worden sind, kann das Ziel für die vorliegende Thematik in einem Satz beschrieben werden: Um bestmögliche Vermittlung von Information und daraus resultierend Bedienbarkeit zu garantieren, ist ein User Interface so zu konstruieren, dass es erstens korrekt und zweitens minimal ist. Vor diesem Hintergrund wird es als nächstes darum gehen, eine konkrete Schnittstelle auf ihre Korrektheit hin zu analysieren, um sich anschließend der Fragestellung zu widmen, inwiefern sich eine korrekte und minimale Bedienungsoberfläche aus einem gegebenen System und seinen Anforderungen formal ableiten lässt.

3 Analyse und Verifikation der Korrektheit von User Interfaces

„even for machines that are seemingly simple [...], finding a correct interface and user-model is not a trivial matter.“

(Heymann & Degani 2002 I, S. 30)

3.1 Beschreibung von Systemen und User Interfaces durch zustandsbasierte Modelle

Wie das vorige Kapitel herausgestellt hat, beschäftigt sich diese Arbeit mit der Beobachtbarkeit der Zustände eines technischen Geräts oder Programms. Innerhalb dieser Problematik lässt sich die Anzeige zur Laufzeit durch Eingabe oder Berechnungen entstandener Daten vernachlässigen. Auch auf die explizite Angabe von *Referenzwerten* kann verzichtet werden. Ein Referenzwert ist ein Parameter, der der internen Logik eines Systems zur Steuerung seines Verhaltens dient (vgl. Degani 2004, S. 96). Beispiele für Referenzwerte sind unter vielen anderen die Systemzeit eines Personal Computers, die Temperatur einer Herdplatte oder die Flughöhe eines Airliners. Zwar kann es von Bedeutung sein, die Schwellen eines solchen Wertes zu kennen, etwa wenn sie zur Erfüllung interner Bedingungen führen, in deren Konsequenz ein Zustandswechsel ausgelöst wird. Das stetige Wissen über genaue Werte ist jedoch für das Beobachten des Zustands im Allgemeinen unerheblich.

Aus diesen Gründen bedarf es hier nicht der Beschreibung von Maschinen und Interfaces in der Weise, wie es eine Turingmaschine leistet. Das heißt, es besteht nicht die Notwendigkeit auf ein „unendliches Gedächtnis“ zurückgreifen zu können. In diesem Fall müsste man nämlich diese Forderung auch an den Benutzer stellen. Vielmehr ist hingegen eine Modellierung durch endliche Automaten in der Form von Zustandsübergangsdiagrammen ausreichend. Im Folgenden wie auch in dem im Rahmen dieser Arbeit entwickelten Software-Tool (vgl. auch Kapitel 5) wird sich dabei auf „flache“ Zustandsautomaten beschränkt; spezielle *superstates* oder *history states,* wie sie in der von David Harel begründeten Notation der *Statecharts* auftauchen (vgl. Harel 1987), werden somit nicht verwendet. Dies stellt keine Verminderung der Aussagekraft dar, denn jedes Statechart-Diagramm lässt sich in einen äquivalenten Zustandsautomaten umwandeln und umgekehrt.

„We consider machines that interact with their human users, interact with the environment, and can act automatically“ (Heymann & Degani 2006, S. 5). Zustände der gerade beschriebenen Diagramme repräsentieren interne Konfigurationen oder Modi solcher Maschinen. Transitionen stehen für diskrete Zustandswechsel, die durch benutzergesteuerte oder automatisch auftretende Ereignisse ausgelöst werden. Automatische Transitionen können durch die Erfüllung interner

Bedingungen getriggert werden oder durch externe Ereignisse der Umgebung, zum Beispiel wenn eine Temperaturänderung dazu führt, dass ein Heizsystem aktiviert oder deaktiviert wird. Im vorliegenden Kontext wird jedoch nicht zwischen diesen Arten von automatischen Ereignissen unterschieden: benutzergesteuerte Transitionen werden als durchgezogene Pfeile, automatische Transitionen als gestrichelte Pfeile dargestellt (VGL. HEYMANN & DEGANI 2002 I, S. 6). Außerdem ist anzumerken, dass die wichtige Frage, welche internen Ereignisse äquivalent sind, einem Problemfeld (dem der Systementwicklung) angehört, das hier außen vor bleiben soll.

Sei nun also das Modell einer Maschine betrachtet. Abbildung 3.1.1 zeigt das unterliegende Verhalten einer Druckknopf-gesteuerten Fußgängerampel. In dem als solchem markierten (beliebig gewählten) Startzustand Drive-1 haben Autos Vorrecht. Betätigen des Knopfes der Ampel – symbolisiert durch das benutzergesteuerte Ereignis ***Press*** – bewirkt einen Wechsel zu Drive-2. Von dort aus wird automatisch (Ereignis *To NOR*) erst nach NOR-1, dann (*To Walk*) nach Walk und anschließend (*To NOR*) nach NOR-2 geschaltet. Während dieses Ablaufs bleibt ein erneutes Drücken des Knopfes ergebnislos, was die mit *Press* gelabelten *self-loops* ausdrücken. Der Folgezustand von NOR-2 ist schließlich Drive-1, es sei denn, *Press* wurde bereits ausgelöst; in diesem Fall wird erst NOR-3 erreicht und danach Drive-2. Walk ist der einzige Zustand, in dem Fußgänger die Straße überqueren dürfen, in NOR-1, NOR-2 und NOR-3 müssen sie genauso wie die Autos warten. Solch ein Modell ist das *Maschinenmodell* eines Systems.

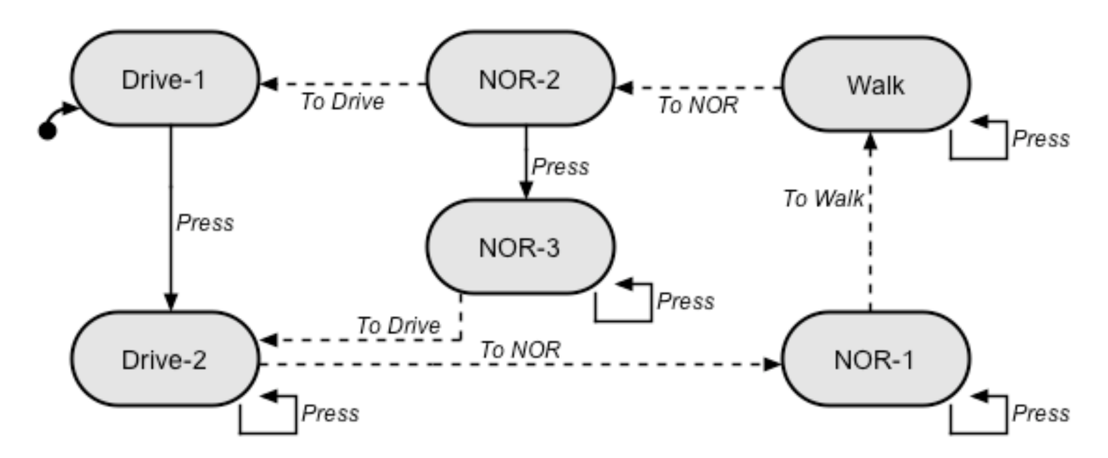


Abbildung 3.1.1: Maschinenmodell einer Druckknopf-gesteuerten Fußgängerampel

Definition 3.1.1: Maschinenmodell / machine model

Das **Maschinenmodell** eines Systems ist ein endlicher Zustandsautomat, der das Verhalten des Systems vollständig beschreibt. Es ist formal definiert durch ein 6-Tupel $(\Sigma^M_{user}, \Sigma^M_{auto}, Q^M, q^M_0, \delta^M_{user}, \delta^M_{auto})$ mit

- endlichen Eingabealphabeten Σ^M_{user} für benutzergesteuerte Ereignisse und Σ^M_{auto} für automatische Ereignisse
- einer endlichen Menge Q^M an Zuständen inklusive eines Startzustands $q^M_0 \in Q^M$
- Übergangsfunktionen $\delta^M_{user}: Q^M \times \Sigma^M_{user} \rightarrow Q^M$ und $\delta^M_{auto}: Q^M \times \Sigma^M_{auto} \rightarrow Q^M$

Eins der wesentlichen Ziele dieser Arbeit ist, Techniken aufzuzeigen, mit denen sich der Weg von der Maschine, repräsentiert durch das Maschinenmodell, hin zum User Interface beschreiten lässt. Eine Schnittstelle enthält in der Praxis eine Steuerungseinheit, mit der der Benutzer die

nötigen Eingaben tätigt, und eine Anzeigeeinheit, durch die dem Benutzer Informationen über das System mitgeteilt werden. Formaler betrachtet besteht ein Interface zusammen mit der Anleitung (etc.) aus einer Auflistung und Beschreibung der für den Benutzer erkennbaren Ereignisse. Da ein Interface eine Abstraktion des unterliegenden Systems verkörpert, beinhaltet dies nur einen Ausschnitt der automatischen Ereignisse und selbige unter Umständen auch maskiert, also als *ein* nach außen getragenes Ereignis (VGL. HEYMANN & DEGANI 2002 I, S. 7F). Dem entgegen gehören natürlich alle benutzergesteuerten Ereignisse dazu. „It is noteworthy that events *per se* cannot be displayed in the interface. What can be displayed is some consequence of their occurrence" (HEYMANN & DEGANI 2002 I, S. 8).
Wie sieht nun das User Interface einer Fußgängerampel aus, die mithilfe eines Knopfes bedient wird? Abbildung 3.1.2 stellt auf der linken Seite die komplette Anzeigeeinheit einer typischen Fußgängerampel dar, der Knopf der Ampel hat also selbst keine weitere Anzeige. In dieser Form sind Fußgängerampeln oft im Straßenverkehr vorzufinden, keineswegs aber immer, worauf noch eingegangen wird. Solche Ampeln sind dafür verantwortlich, dass Leute häufig zahlreiche Male auf den Knopf drücken, um sicher zu gehen – oder besser: die *Chance* zu erhöhen –, dass das System den Wunsch, die Straße zu überqueren, registriert hat.

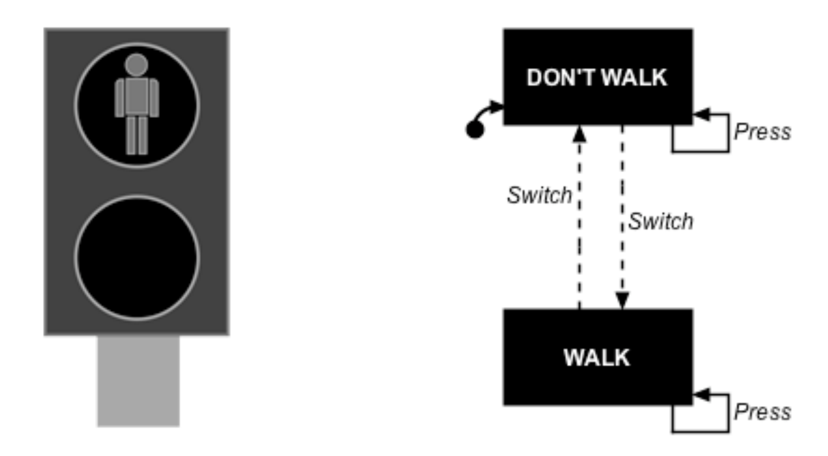


Abbildung 3.1.2: Anzeigeeinheit einer Druckknopf-gesteuerten Ampel (links) und das zur Ampel gehörige Oberflächenmodell (rechts)

Was vermittelt dieses User Interface an Information? Die rechte Seite von Abbildung 3.1.2 zeigt das Modell des Interfaces, welches verdeutlicht, wie der Benutzer die Zustände des Systems wahrnehmen kann: Im Zustand DON'T WALK (rotes stehendes Männchen) lässt sich zwar der Knopf der Ampel drücken (*Press*), bleibt jedoch vorerst ohne sichtbares Resultat. Irgendwann führt das automatische Ereignis *Switch* dazu, dass der Zustand WALK (grünes laufendes Männchen) aktiviert wird. Auch hier kann der Knopf betätigt werden, was dieses Mal gar jeglicher Wirkung entbehrt, wie sich mit Blick auf das Maschinenmodell in Abbildung 3.1.1 erkennen lässt. Erst ein erneutes Auftreten von *Switch* ändert wieder den Zustand, führt nämlich zum Zurückschalten der Ampel von Grün zu Rot (für Fußgänger). Die Bezeichnung *Switch* wurde hier exemplarisch ausgesucht, um die Maskierung interner Ereignisse zu demonstrieren. Es handelt sich bei dem abgebildeten Diagramm um das so genannte *Oberflächenmodell* eines Interfaces – im Englischen: *user model* –, das wie folgt formal definiert ist:

Definition 3.1.2: Oberflächenmodell / user model

Das **Oberflächenmodell** eines Systems ist ein endlicher Zustandsautomat, der die einem User Interface inklusive aller zugehörigen Hilfsmaterialien entnehmbaren Informationen beschreibt. Es ist formal definiert durch ein 6-Tupel $(\Sigma_{user}, \Sigma^U_{auto}, Q^U, q^U_0, \delta^U_{user}, \delta^U_{auto})$ mit

- endlichen Eingabealphabeten Σ_{user} für benutzergesteuerte Ereignisse und Σ^U_{auto} für (ggf. maskierte) automatische Ereignisse
- einer endlichen Menge Q^U an Zuständen inklusive eines Startzustands $q^U_0 \in Q^U$
- Übergangsfunktionen $\delta^U_{user}: Q^U \times \Sigma_{user} \rightarrow Q^U$ und $\delta^U_{auto}: Q^U \times \Sigma^U_{auto} \rightarrow Q^U$

Es handelt sich bei Σ_{user} um dieselbe Menge wie die des Maschinenmodells. Der Zusammenhang zwischen den Eingabealphabeten Σ^M_{auto} und Σ^U_{auto} wird in Kapitel 3.3 genauer charakterisiert.
Wie bereits implizit gesagt, erhält der Benutzer bei dem gegebenen User Interface der Ampel nicht genügend Rückmeldung, um sich jederzeit des Zustands des Systems bewusst zu sein. Für das Wissen über den Erfolg seiner Aktion (das Drücken des Knopfes) sollte es also Feedback geben. Aus diesem Grund haben gerade in den letzten Jahren viele solcher Fußgängerampeln oberhalb von Rot und Grün oder eingebettet in den Druckknopf eine weitere Anzeige erhalten, die beispielsweise einen Text wie „Grün kommt“ ausgibt. Aber resultiert daraus schon ein korrektes Interface? Sofern nicht gerade Autos direkt vor der Ampel plötzlich anhalten, weiß der Fußgänger nicht, wann es zu dem Wechsel von DON'T WALK zu WALK kommt. An manchen Kreuzungen findet sich eine Anzeige, die ein gelb blinkendes laufendes Männchen zeigt. Dies dient jedoch eher der Warnung für Autofahrer. Denn für ein korrektes Interface ist es nicht ausschlaggebend, dass der Benutzer im Vorhinein weiß, *wann* es zu einem Ereignis kommen kann, sondern nur *dass* es dazu kommen kann. Drei Zustände genügen somit der Eigenschaft der Korrektheit, und zwar so, wie sie im folgenden Oberflächenmodell zusammengefasst sind (Abbildung 3.1.3), das mit dem nun vorhandenen Vorwissen keiner weiteren Erklärung bedarf. Eine Beschäftigung mit phonetischen Signalen, die eine Fußgängerampel sehbehindertengerecht macht, soll hier nebenbei nicht stattfinden, denn es geht ja nur darum, *welche* Informationen angezeigt werden, nicht *auf welche Weise* diese „angezeigt“ werden.

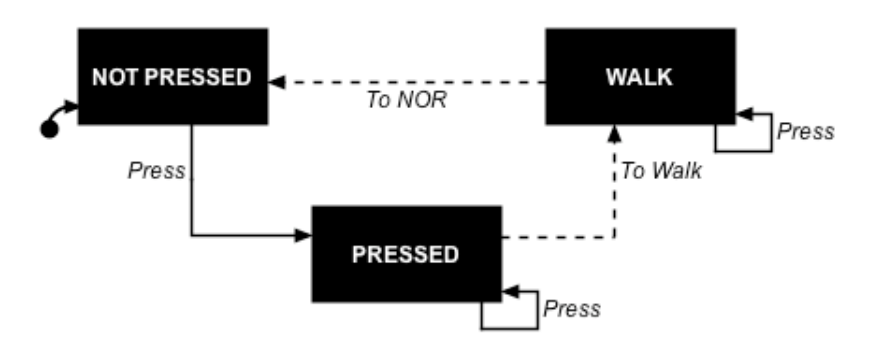


Abbildung 3.1.3: Korrektes Oberflächenmodell einer Druckknopf-gesteuerten Fußgängerampel

Die Erarbeitung des Oberflächenmodells wurde hier noch intuitiv ohne formale Überprüfung der Korrektheit der Ergebnisse durchgeführt. In diesem Kapitel wird daher ein Verfahren vorgestellt, mit dem sich ein Oberflächenmodell bezüglich des zugehörigen Maschinenmodells verifizieren lässt. Das Vorgehen wird an einem gegenüber der Ampel etwas komplexeren Beispiel illustriert werden, das insbesondere später bei der automatischen Generierung mehr Erkenntnisse

liefern wird, aber auch schon hier die Notwendigkeit einer Verifikationsmethode verdeutlicht. Um diese Methode anwenden zu können, muss jedoch eine Möglichkeit bestehen, die Spezifikation der Aufgaben, die an ein System in Bezug auf die Beobachtbarkeit gestellt werden, zu formalisieren. Ein Konzept, das diesem Anspruch gerecht zu werden versucht, wird deshalb zunächst noch anhand des Beispiels der Fußgängerampel eingeführt.

3.2 SPEZIFIKATIONSKLASSEN ALS FORMALISIERUNG DER AUFGABENANFORDERUNG

Michael Heymann und Asaf Degani widmen sich in den genannten Papern der Rolle der Aufgaben des Benutzers nur bedingt (vgl. Heymann & Degani 2006, S. 6f sowie Heymann & Degani 2002 I, S. 7), aber es sei schon zu Beginn erwähnt, dass der von ihnen verfolgte Ansatz eine Einschränkung der beiden Verfahren darstellt.
Im Zuge der Beobachtbarkeit eines automatisierten Systems hat ein Benutzer das Ändern von Modi zu verfolgen, ebenso wie er selbst Eingabesequenzen tätigen und die Konsistenz zwischen dem Weg zum gewünschten Ziel und der aktuellen Situation überwachen muss. Ein Modus ist dabei nicht zwangsläufig ein genauer Zustand des Systems, sondern repräsentiert in der Regel viel häufiger mehrere interne Zustände. Die Anforderung an das User Interface, mit der sich an dieser Stelle beschäftigt werden soll, ist die Gewährleistung der Beobachtbarkeit der Modi des unterliegenden Systems; „[...] a common task specification in an automated control system is that the user be able to determine unambiguously the current and the subsequent mode of the machine“ (Heymann & Degani 2002 I, S. 7). Heymann und Degani formalisieren solche Modi in Form so genannter *Spezifikationsklassen,* die sie wie folgt definieren:

Definition 3.2.1: Spezifikationsklasse / specification class
(vgl. Heymann & Degani 2006, S. 6)

> Eine **Spezifikationsklasse** innerhalb eines Systems ist eine Menge interner Zustände, zwischen denen der Benutzer inhaltlich nicht unterscheiden können muss. Jeder Zustand des Systems ist Element genau einer Spezifikationsklasse.
> Sei M die Menge der Zustände und C die Menge der Spezifikationsklassen eines Systems. Dann gilt: $\forall S \in M: (\exists C_i \in C: (S \in C_i) \wedge \forall C_j \in C: (i \neq j \Rightarrow s \notin C_j))$

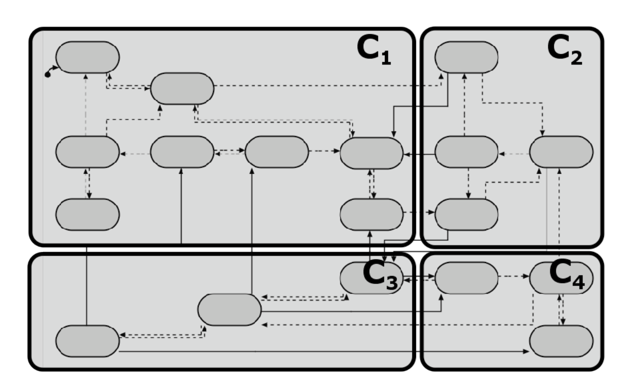


Abbildung 3.2.1: Einteilung der Zustände eines Maschinenmodells in Spezifikationsklassen

Die Einteilung der Spezifikationsklassen entspricht also einer disjunkten Clusterung der Zustände eines Systems, verbildlicht durch Abbildung 3.2.1. Es handelt sich dabei folgerichtig um eine Partition, Zustände einer Klasse heißen spezifikationsäquivalent. Die Entscheidung, welche Zustände zu einer Klasse zusammengefasst werden, wird von Design-Teams mittels Anwendung von Techniken der Aufgabenanalyse sowie Experteneingaben vorgenommen. Dieser Schritt der Erzeugung eines Interfaces erfordert also sehr wohl menschlichen Einfluss, lässt sich aber in den generellen Entwicklungsprozess einer Software integrieren, wie ihn etwa die *Unified Modeling Language (UML)* vorsieht (VGL. HEYMANN & DEGANI 2006, S. 6F UND S. 24F).

Hinweis: Bei der Auseinandersetzung mit den Eigenschaften von Spezifikationsklassen wurde klar, dass Definition 3.2.1, die sich auf die Informationen stützt, welche den Quellen DEGANI & HEYMANN 2002, HEYMANN & DEGANI 2002 I-III sowie HEYMANN & DEGANI 2006 zu entnehmen sind, unzureichend für das in Kapitel 4 beschriebene Verfahren zur automatischen Generierung von Interfaces ist. Im Anschluss daran wird die Definition daher noch einmal untersucht werden und es wird sich zeigen, dass eine weitere Bedingung für die Einteilung in Spezifikationsklassen gefordert werden muss.
Die Betrachtung wird jedoch erst dann sinnvoll, wenn der Generierungsalgorithmus hinreichend bekannt ist. Für den Moment reichen außerdem die oben stehenden Angaben aus, um das Klassenkonzept zu verstehen.

Zur Veranschaulichung der Bedeutung von Spezifikationsklassen sei erneut das Maschinenmodell der Fußgängerampel untersucht. Abbildung 3.2.2 zeigt eine Einteilung der sechs Zustände in drei verschiedene Modi, die auf den Anforderungen eines Fußgängers basiert. Der Benutzer soll einerseits erkennen können, ob er die Straße passieren darf. Deswegen befindet sich der Zustand Walk getrennt von allen anderen Zuständen in der Klasse **WALK**. Andererseits muss Feedback bezüglich der Benutzeraktion *Press* gegeben werden: das geschieht durch die Aufteilung in die Klassen **NOT PRESSED** und **PRESSED**, so dass der Benutzer immer weiß, ob das Drücken des Knopfes zum gewünschten Ergebnis geführt hat.

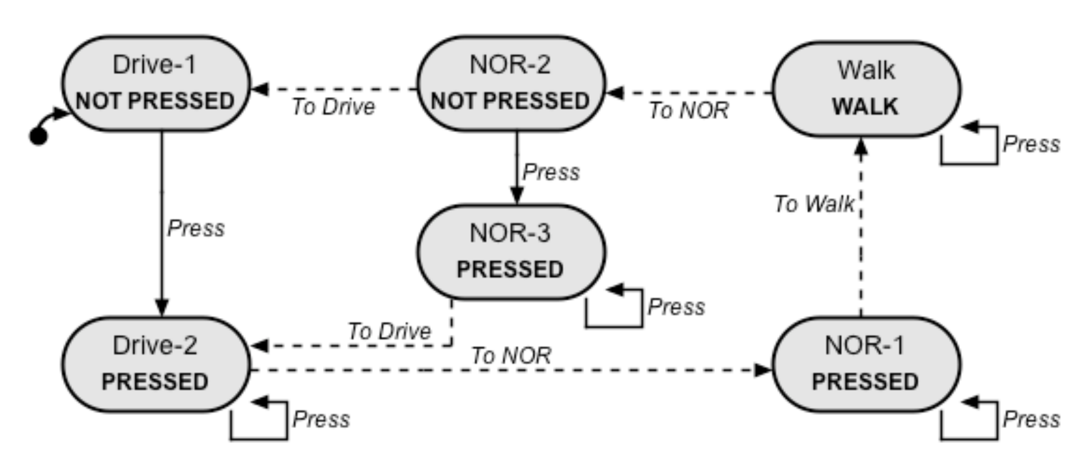


Abbildung 3.2.2: Maschinenmodell der Fußgängerampel samt Spezifikationsklassen aus Sicht der Fußgängers

Bei Vergleich mit dem als korrekt *vermuteten* Oberflächenmodell aus Abbildung 3.1.3 ist festzustellen, dass die Zustände jeder Klasse anscheinend jeweils komplett zusammengezogen werden können. Das ist aber keineswegs der Regelfall, wie in Kapitel 3.3 ersichtlich sein wird.

Außer vielleicht an einer Kreuzung entspricht im Allgemeinen nun das Verhalten des Systems einer Fußgängerampel im Grunde dem der zugehörigen Ampel für Autos. Die beiden sind unmittelbar miteinander gekoppelt, lassen sich letzten Endes als *ein* zusammenhängendes System begreifen. Das bekannte Modell der Maschine bezieht sicht also auch nicht nur auf die Fußgängerampel, sondern vielmehr auf das Ampelsystem insgesamt. Ein Unterschied ist jedoch, dass das für den Fußgänger als benutzergesteuert geltende Ereignis *Press* im Maschinenmodell der Fahrzeug-Ampel zu den automatischen Ereignissen zählt, denn ein Autofahrer kann dieses nicht beeinflussen. Darüber hinaus muss zur Konstruktion des User Interfaces für Pkws vor allem eine andere Zusammensetzung der Klassen gewählt werden, wie sie in Abbildung 3.2.3 zu sehen ist.

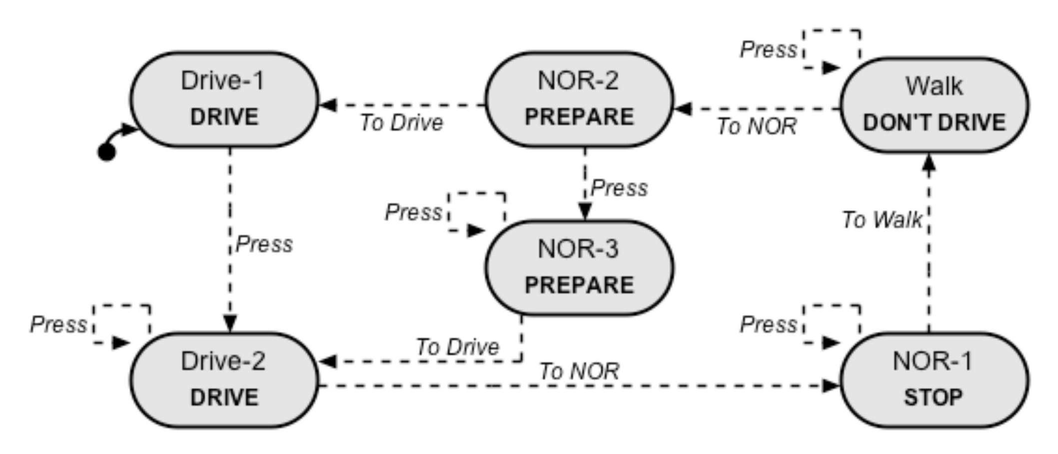


Abbildung 3.2.3: Maschinenmodell der Fußgängerampel samt Spezifikationsklassen aus Sicht des Autofahrers

Es handelt sich zwar hierbei um eine Maschine, die der Benutzer rein beobachtend wahrnimmt, doch liegen natürlich auch solche Systeme im Blickpunkt dieses Kontexts. Offensichtlich können die vier Klassen den Farben einer Ampel zugeordnet werden: **DON'T DRIVE** zu rot, **DRIVE** zu grün, **STOP** zu gelb sowie **PREPARE** zu rot/gelb. Das resultierende Interface lässt sich folglich leicht erahnen und sei an dieser Stelle vernachlässigt.

Wichtiger ist die Schlussfolgerung, dass die Zustände, die einem Benutzer anzuzeigen sind, damit dieser das zugehörige System korrekt bedienen (im letzten Beispiel lediglich: beobachten) kann, von der Aufgabe, für die das System entwickelt wurde, in hohem Maße abhängig ist. Die Frage, welche Zustände spezifikationsäquivalent sind, ist sicherlich bei umfangreichen Modellen nicht immer leicht zu beantworten. Anhand der Fußgängerampel wurde weiterhin gezeigt, dass die Einteilung in Spezifikationsklassen eine wesentliche Vorarbeit bedeutet. Die Behandlung der Entstehung der Klassen wird in dieser Arbeit aber nicht von fortwährendem Interesse sein. Vielmehr wird im weiteren Lauf davon ausgegangen werden, dass das Maschinenmodell und die Aufgabenspezifikation gegeben sind. Damit sind alle Vorkehrungen getroffen, um die von Heymann und Degani entwickelte Methode zur Verifikation eines Interfaces an einem Beispiel zu erläutern, das immerhin neun Zustände und 24 Transitionen aufweist: einem halbautomatischen Heiz- und Kühlsystem.

3.3 VERFAHREN ZUR VERIFIKATION DES USER INTERFACES EINER MASCHINE

Nachdem die Modellierung von Systemen durch zustandsbasierte Modelle erörtert und eine Möglichkeit, die Spezifikation der Aufgaben eines Benutzers zu formalisieren, aufgezeigt wurde, wird in diesem Teilkapitel die Korrektheit potentieller User Interfaces (beschrieben durch ihre assoziierten Oberflächenmodelle) bezüglich des Maschinenmodells des unterliegenden Systems überprüft. Die dabei verwendete Methode wurde von Heymann und Degani entwickelt und dient dazu, *error states, restricting states* und *augmenting states* aufzuspüren. Sie soll an einem neuen Beispiel, das auch für die Vermittlung des Verfahrens zur Generierung von Interfaces nützlich sein wird, veranschaulicht werden, welches im Folgenden vorgestellt wird.
Abbildung 3.3.1 zeigt das Maschinenmodell eines halbautomatischen Heiz- und Kühlsystems, das bereits die Spezifikation dreier für den Benutzer zu unterscheidenden Modi enthält: **COOL**, **WARM** und **HOT**. Jede dieser Spezifikationsklassen umfasst wiederum drei interne Zustände, zwischen denen das System bei Bedarf automatisch hin- und herwechselt. Grund für diese Wechsel sind externe Temperaturschwankungen, auf die das System reagiert, was hier aber nicht von Interesse sein wird. Hingegen ist wichtig, dass für das Triggern einer Transition innerhalb von **COOL** sowie zwischen Warm-A und Warm-B die Ereignisse *cool+* und *cool-* verantwortlich sind, während zwischen Warm-B und Warm-C genauso wie in **HOT** eine Zustandsänderung das vorherige Auftreten von *hot+* oder *hot-* impliziert. Der Benutzer kann die gewünschte Stufe des Systems durch zwei Steuerungselemente einstellen, die das Schalten der entsprechenden Transitionen bewirken: *heat up* führt – wenn möglich – zu einem Erwärmen der Umgebung, *cool down* verringert andersherum die Temperatur.

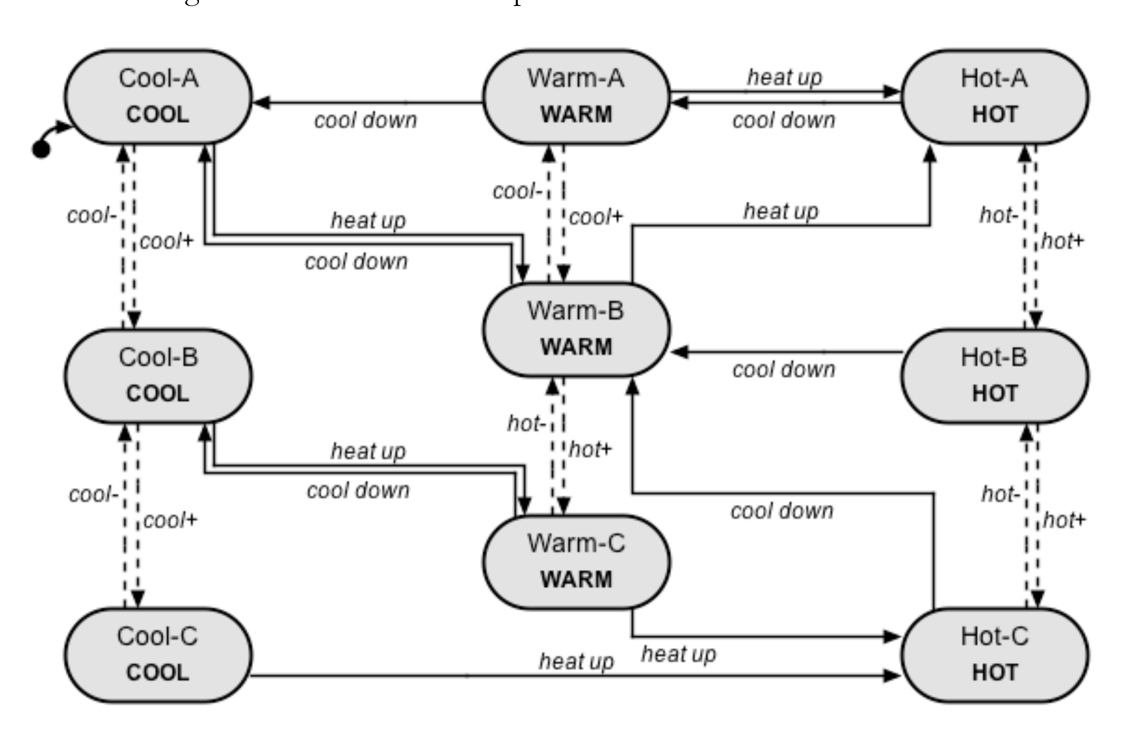


Abbildung 3.3.1: Maschinenmodell eines halbautomatischen Heiz- und Kühlsystems inklusive Spezifikationsklassen

Es ist nun ein Interface erforderlich, das dem Benutzer erlaubt, den aktuellen und nächsten Modus des Heiz- und Kühlsystems in Abhängigkeit zur Eingabe zu jedem Zeitpunkt erkennen zu

können. Bei der Ampel wurde argumentativ angenommen, dass für die drei Klassen jeweils genau eine Anzeige vorhanden sein muss. Dementsprechend wäre auch für das vorliegende Beispiel ein intuitiver Ansatz, das Interface so zu konstruieren, dass es dieser Bedingung gerecht wird. In Abbildung 3.3.2 ist eine simple und gut überschaubare solche Schnittstelle dargestellt: Sie zeigt die aktive Stufe (hier: „hot") dunkel hinterlegt an, die anderen hell. Durch die Tasten „cool down" und „heat up" lässt sich die Stufe ändern.

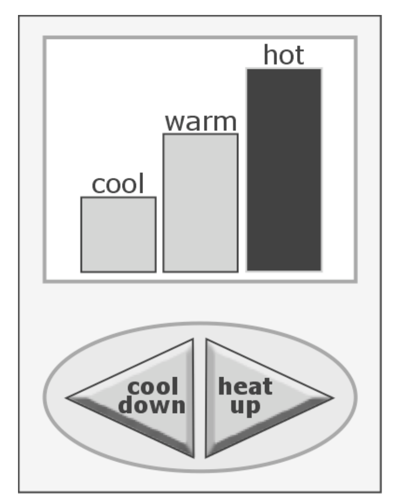


Abbildung 3.3.2: Potentielles User Interface für das Heiz- und Kühlsystem

Das Oberflächenmodell dieses User Interfaces beinhaltet offensichtlich drei Zustände; welche Transitionen verlaufen aber zwischen diesen? Im Maschinenmodell in Abbildung 3.3.1 ist zu sehen, dass fast alle benutzergesteuerten Transitionen jeweils zur daneben befindlichen Klasse führen. Eine Ausnahme bildet lediglich die von Cool-C ausgehende Transition *heat up*, durch die das System direkt in den Zustand Hot-C gelangt. Die Schnittstelle selbst gibt keinen Aufschluss darüber, dass das Betätigen der Taste „heat up" in der Stufe „cool" einen Wechsel nach „hot" als Konsequenz haben kann. Je nach dem, ob diese Situation in einer zugreifbaren Bedienungsanleitung dargelegt ist oder nicht, ergibt sich daher entweder das in Abbildung 3.3.3 oder das in Abbildung 3.3.4 dargestellte Oberflächenmodell. Beide blenden alle internen Ereignisse vollständig aus.



Abbildung 3.3.3: Oberflächenmodell des Heiz- und Kühlsystems ohne externe Zusatzinformationen



Abbildung 3.3.4: Oberflächenmodell des Heiz- und Kühlsystems mit externen Zusatzinformationen

Wie dem auch sei, ohne Anwendung eines speziellen Verfahrens lässt sich erkennen, dass beide Oberflächenmodelle keine korrekte Bedienbarkeit ermöglichen. Ersteres macht falsche Aussagen über die möglichen Zustandswechsel, verschweigt genauer gesagt den Übergang von COOL nach HOT. Damit konstituiert COOL einen *restricting state*. Bei zweiterem liegt auf der Hand, dass es ein nicht-deterministisches Interface repräsentiert, der Benutzer kann im Zustand COOL nicht vorhersagen, welcher Folgezustand aus dem Drücken von „heat up" resultiert. Es kann somit geschlossen werden, dass das User Interface aus Abbildung 3.3.2 sowie das zugehörige Modell der Oberfläche die in Kapitel 2.2 aufgeführten Korrektheitskriterien nicht erfüllt.
Die beiden Modelle liefern dennoch eine wichtige Beobachtung mit Blick auf das Interface: „The user model is based on the interface because it directly relates to the indications displayed there. Thus, [...] the interface is actually embedded in the user model" (HEYMANN & DEGANI 2006, S. 8). Das heißt, für alle folgenden Betrachtungen kann das konkrete Design einer Schnittstelle übergangen werden, die Beschäftigung mit dem Oberflächenmodell deckt dieses vollkommen ab. Außerdem enthält es implizit Aussagen, die etwa eine Bedienungsanleitung auflistet, und zwar in Form der Transitionen. Eine Transition β zwischen zwei Zuständen S_1 und S_2 besagt ja schließlich nichts anderes als: Wenn das System in Zustand S_1 ist und Ereignis β ausgelöst wird, so schaltet das System in den Zustand S_2. Aus diesem Grund kann also auch auf die Unterscheidung dazwischen verzichtet werden, welche Informationen ein User Interface selbst preisgibt und welche sich in den beigefügten Hilfsmaterialien befinden (VGL. HEYMANN & DEGANI 2006, S. 8). Das Oberflächenmodell verbindet all diese Quellen (siehe auch Definition 3.1.2) und stellt sie abstrakt dar, weshalb es für die vorliegenden Zwecke überlegen ist.

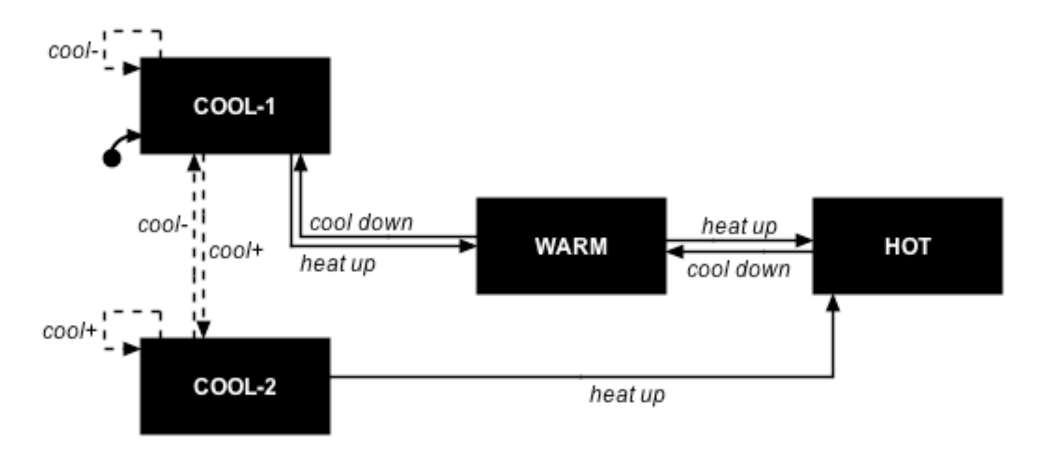


Abbildung 3.3.5: Alternatives Oberflächenmodell des Heiz- und Kühlsystems

Das Problem bei den oben abgebildeten Modellen entstand aus den unterschiedlichen internen Folgezuständen von Cool-A und Cool-B gegenüber Cool-C bezüglich der Transition *heat up*. Das Modell in Abbildung 3.3.5 versucht diese Schwachstelle nun zu beheben, indem es den Modus **COOL** in zwei Untermodi aufteilt, die genau auf die Differenz zwischen den Zuständen in **COOL**, was eingehende und ausgehende Transitionen betrifft, abzielen. Dies zeigt ein Vergleich mit dem Maschinenmodell in Abbildung 3.3.1: Triggern von *cool down* hat aus allen Zuständen in **WARM** einen der Zustände Cool-A und Cool-B als Ergebnis und umgekehrt führt *heat up* von dort aus, anders als bei Cool-C, immer in den Modus **WARM**. Daher werden Cool-A und Cool-B durch COOL-1, Cool-C hingegen durch COOL-2 repräsentiert. Außerdem gewährt dieses Oberflächenmodell einen partiellen Einblick in die Menge automatischer

Ereignisse, bildet nämlich ab, dass interne Wechsel zwischen COOL-1 und COOL-2 vorkommen können.
Aber handelt es sich hierbei jetzt um ein korrektes User Interface? Diese Frage ist allein durch genaues Hingucken schon hier schwer zu beantworten, bei steigender Komplexität der Systeme wird solch ein Vorgehen zweifelsohne (nahezu) unmöglich. Es bedarf einer Methode, die die Korrektheit eines Oberflächenmodells zu einem gegebenen Maschinenmodell und der formalen Aufgabenspezifikation anhand definierter Abläufe verifiziert. Solch eine Methode wird auf den folgenden Seiten auf formalem und graphischem Weg erarbeitet und durchgeführt.

„In any human-machine system, two concurrent processes are constantly at play: the machine with its internal states and transitions on the one hand, and the interface annunciations with the associated user-model transitions on the other" (HEYMANN & DEGANI 2006, S. 10). Diese beiden Prozesse, repräsentiert durch das Maschinen- und das Oberflächenmodell, müssen stets synchron bleiben, inkonsistente Zustände also vermieden werden. Genauer gesagt, das Oberflächenmodell darf weder *error states* noch *restricting states* noch *augmenting states* aufweisen. Es ist also gefordert, dass der im Interface angezeigte Zustand nach jeder beliebigen, möglichen Ereignissequenz $\beta_1...\beta_k$ konform mit dem internen Zustand des Systems ist.
Das Verfahren dient nun dazu, ein *zusammengesetztes Modell* zu erstellen, das die Abbildung aller möglichen Ereignissequenzen gleichzeitig für das Maschinen- und das Oberflächenmodell leistet. Anhand der dabei entstehenden Zustandspaare (bestehend aus System- und Interface-Zustand) lässt sich beurteilen, ob einer der fehlerhaften Zustandstypen vorliegt. Wenn ein Zustandspaar existiert, bei dem der Zustand des Systems einer anderen Spezifikationsklasse angehört als der des User Interfaces, so stellt dies einen *error state* dar. Findet sich eine ausgehende Transition des Zustands des Maschinenmodells, die einen Wechsel der Spezifikationsklasse bewirkt und die für den zugehörigen Zustand des Oberflächenmodells nicht definiert ist, so begründet der Interface-Zustand einen *restricting state*, im umgekehrten Fall handelt es sich handelt es sich entsprechend um einen *augmenting state*.

Definition 3.3.1: Zusammengesetztes Modell / composite model

Das **zusammengesetzte Modell** eines Systems ist ein endlicher Zustandsautomat, der die gleichzeitige Abbildung aller möglichen Ereignissequenzen eines Maschinenmodells und des zugehörigen Oberflächenmodells darstellt. Es ist formal definiert durch ein 6-Tupel $(\Sigma_{user}, \Sigma^U_{auto}, Q^C, q^C_0, \delta^C_{user}, \delta^C_{auto})$ mit

- endlichen Eingabealphabeten Σ_{user} für benutzergesteuerte Ereignisse und Σ^U_{auto} für (ggf. maskierte) automatische Ereignisse
- einer endlichen Menge $Q^C \subseteq Q^M \times Q^U$ an Zustandspaaren bestehend aus System- und Interface-Zustand inklusive eines Startzustands $q^C_0 = (q^M_0, q^U_0) \in Q^C$
- Übergangsfunktionen $\delta^C_{user}: Q^C \times \Sigma^C_{user} \rightarrow Q^C$ und $\delta^C_{auto}: Q^C \times \Sigma^C_{auto} \rightarrow Q^C$

Bevor die Entwicklung eines solchen Modells graphisch veranschaulicht stattfinden soll, seien zunächst die Eingabealphabete Σ^M_{auto} und Σ^U_{auto} des Maschinen- und des Oberflächenmodells eingehender untersucht. Dies wird auch erkenntlich machen, wieso das Eingabealphabet des zusammengesetzten Modells für automatische Ereignisse das des Oberflächenmodells ist.

Wie das Konzept der Spezifikationsklassen vorgibt, muss der Benutzer nicht exakt wissen, in welchem Zustand sich das System befindet, sondern nur, welcher Modus aktiv ist und welcher der folgende sein wird. Deshalb können einige interne Ereignisse beim User Interface vernachlässigt werden, andere lassen sich gruppieren. Diese Abstraktion wird durch eine Projektion $\Pi: \Sigma^M_{auto} \rightarrow \Sigma^U_{auto}$ erreicht, die entscheidet, welche Ereignisse im Eingabealphabet des Oberflächenmodells enthalten sind. Die Abbildungsvorschrift kann durch Aufteilung von Σ^M_{auto} in drei disjunkte Teilmengen Σ^M_o, Σ^M_m und Σ^M_u beschrieben werden: Σ^M_o bezeichnet die Menge beobachteter Ereignisse *(observed events)*, welche tatsächlich auch im Oberflächenmodell sichtbar sind, Σ^M_m weiter die Menge maskierter Ereignisse *(masked events)*. Wie bereits angesprochen, werden diese gruppiert mit anderen zusammen als ein Ereignis im Oberflächenmodell aufgeführt. Schließlich sind die unbeobachteten Ereignisse *(unobserved events)* in Σ^M_u diejenigen, welche das Oberflächenmodell verschweigt. Das Eingabealphabet Σ^U_{auto} besteht also aus $\Pi(\Sigma^M_o) = \Sigma^M_o$ und der Projektion $\Pi(\Sigma^M_m)$, die die durch Maskierung ermittelten Ereignisse verkörpert. Außerdem kann das leere Wort ε als Repräsentant für die unbeobachteten Ereignisse angesehen werden. Ein System wird also durch Ereignisse aus Σ^M_{auto} automatisch beeinflusst, während der Benutzer dies durch die angezeigten Ereignisse aus Σ^U_{auto} verfolgen kann. Die Zustandsentwicklungen des Systems und des User Interfaces verlaufen somit parallel. Sie sind jedoch nicht vollends synchronisiert, denn das Interface liefert nur einen Ausschnitt aus dem Verhalten des Systems (VGL. HEYMANN & DEGANI 2002 I, S. 12F).

Das zusammengesetzte Modell wird nun so konstruiert, dass es die parallel verlaufenden Prozesse schrittweise nachvollzieht. Beginnend bei den beiden Startzuständen werden für jedes Zustandspaar anhand der dort definierten ausgehenden Transitionen die nachfolgenden Paare ermittelt. Um etwaige Fehler des Oberflächenmodells aufdecken zu können, sind die Übergänge des Modells mit Ereignissen aus Σ^U_{auto} gelabelt, denn die Situation wird aus Sicht des Benutzers betrachtet. Sei etwa angenommen, das System ist im Zustand S, das Interface im entsprechenden Zustand T. Im Maschinenmodell führe eine automatische Transition β von S zu einem Zustand S'. Wenn β ein beobachtetes Ereignis ist, also $\Pi(\beta) = \beta$ gilt, so muss auch im Oberflächenmodell eine Transition von T zu einem Zustand T' vorhanden sein. Daher enthält das zusammengesetzte Modell eine mit β gelabelte Transition von (S, T) nach (S', T'), wie Abbildung 3.3.6 verdeutlicht. Ist β hingegen maskiert, so handelt es sich bei $\Pi(\beta)$ um das maskierte Abbild von β. Das Oberflächenmodell beinhaltet einen Übergang $\Pi(\beta)$ von T nach T', ebenso führt also wieder im zusammengesetzten Modell (Abbildung 3.3.7) eine Transition $\Pi(\beta)$ von (S, T) nach (S', T').

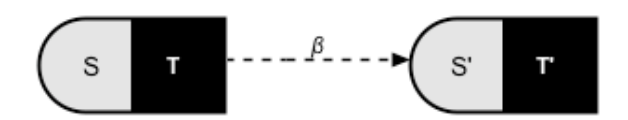


Abbildung 3.3.6: Beobachtetes Ereignis im zusammengesetzten Modell

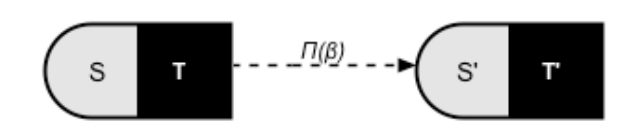


Abbildung 3.3.7: Maskiertes Ereignis im zusammengesetzten Modell

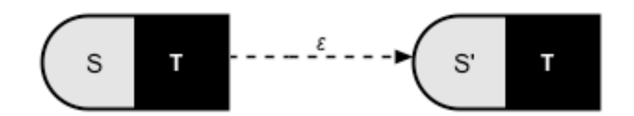


Abbildung 3.3.8: Unbeobachtetes Ereignis im zusammengesetzten Modell

Falls jedoch β als letzte Möglichkeit unbeobachtet ist, dann wird in T kein Zustandswechsel ausgelöst, der zu T' schaltet. Der Folgezustand von (S, T) ist also (S', T), denn für den Benutzer ändert sich im Moment des Auftretens des Ereignisses β nichts. Das Label ε drückt dies aus, sichtbar in Abbildung 3.3.8 (VGL. HEYMANN & DEGANI 2002, S. 13). Mit diesen Vorkenntnissen kann nun das *verbesserte* Modell der Oberfläche (siehe Abbildung 3.3.5) des Heiz- und Kühlsystems aus Abbildung 3.3.1 analysiert werden.
Die Konstruktion des zusammengesetzten Modells des Heiz- und Kühlsystems startet beim Zustandspaar (Cool-A, COOL-1), da dies der Ausgangspunkt von Maschine und User Interface ist. Das benutzergesteuerte Ereignis *heat up* bringt das System in den Zustand Warm-B, passend dazu wird im Interface der Modus WARM aktiv. Außerdem kann in (Cool-A, COOL-1) das beobachtete Ereignis *cool+* stattfinden, das im Folgezustandspaar (Cool-B, COOL-2) resultiert. Abbildung 3.3.9 veranschaulicht diesen ersten Schritt bei der Entwicklung des Modells.

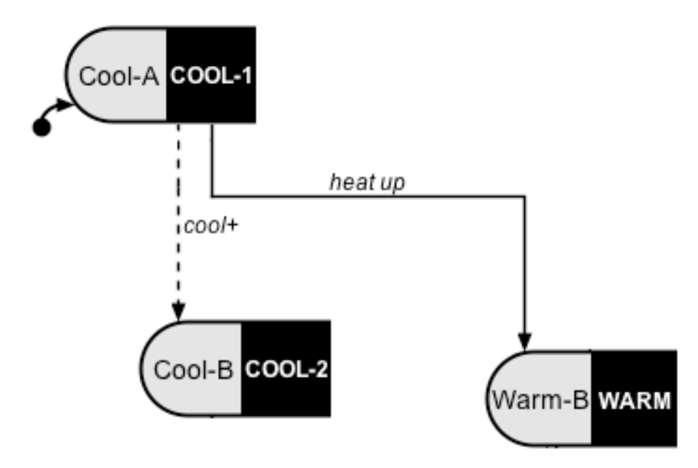


Abbildung 3.3.9: Ein Schritt der Konstruktion des zusammengesetzten Modells

Der weitere Aufbau des gesamten Diagramms sei hier nicht explizit beschrieben, es werden einfach nach und nach alle ausgehenden Transitionen aller auftretenden Zustandspaare überprüft und im Diagramm umgesetzt. Es ergibt sich das auf der folgenden Seite abgebildete, vollständige zusammengesetzte Modell (Abbildung 3.3.10). Dort zeigt sich, dass auch dieser Versuch der Angabe eines korrekten Oberflächenmodells gescheitert ist: Die Transition *heat up* führt vom Zustandspaar (Cool-B, COOL-2) aus in den grau hinterlegten inkonsistenten Zustand (Warm-C, HOT). Das User Interface besagt offensichtlich, dass das System sich im Modus HOT befindet, obwohl es tatsächlich in einem Zustand der Spezifikationsklasse **WARM** ist. Es handelt sich bei solch einem Zustand um einen *error state.* „Given such a display, there is nothing we can do to alleviate the problem; no additional training, no better user manual, procedures, or any other countermeasures will help" (HEYMANN & DEGANI 2006, S. 11). Es folgt, dass auch dieser Vorschlag für ein User Interface inkorrekt ist.

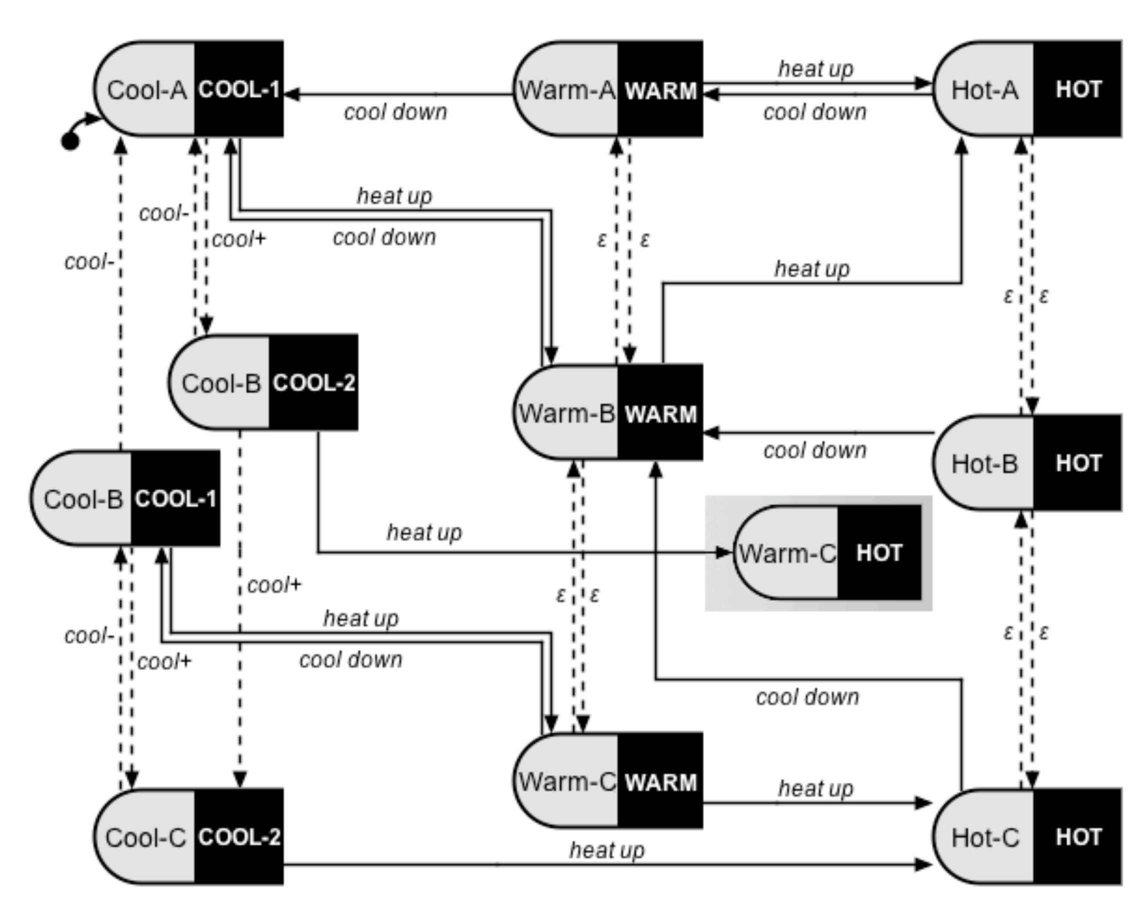


Abbildung 3.3.10: Zusammengesetztes Modell des Heiz- und Kühlsystems, das einen error state (grau hinterlegt) aufweist

Natürlich ließe sich das bis hierher verfolgte Vorgehen zum Finden eines korrekten Interfaces weiterführen, indem solange neue Oberflächenmodelle verifiziert werden, bis kein Fehler mehr gefunden wird. Ein solcher Ansatz ist aber nicht nur ineffizient und bei höherer Komplexität der Systeme im schlechten Fall praktisch endlos, sondern er mündet vor allem auch nicht in dem gewünschten Ziel, ein Interface zu entwickeln, das sowohl korrekt als auch minimal ist: „[...] even when a correct interface is found, there is no assurance that it is the simplest" (HEYMANN & DEGANI 2002 I, S. 15). Es muss also ein systematisches Verfahren gefunden werden, das die korrekte und gleichzeitig minimale Informationsvermittlung garantiert. Michael Heymann und Asaf Degani haben einen Algorithmus entworfen, der versucht, dieser Forderung gerecht zu werden. Er wird Thema des folgenden Kapitels und letztlich der Kernaspekt dieser Arbeit sein.

Auf Beispiele zur methodischen Ermittlung von *restricting states* und *augmenting states* wird an dieser Stelle nebenbei verzichtet. Die Überprüfung der Korrektheit des gleich vorgestellten Verfahrens, welche eine erneute Beschäftigung mit dem Konzept der Spezifikationsklassen erfordern wird, wird jedoch die Entdeckung eines *augmenting states* in einem abstrakten Beispiel hervorbringen. Ein *restricting state* wurde ja bereits zu Beginn dieses Teilkapitels beim ersten Oberflächenmodell ausfindig gemacht, doch auch *restricting states* werden noch im Blickpunkt dieser Arbeit stehen. Im zusammengesetzten Modell würde ein solcher Zustand eine unbeobachtete ausgehende benutzergesteuerte Transition aufweisen.

4 GENERIERUNG VON KORREKTEN UND MINIMALEN OBERFLÄCHENMODELLEN

„[...] as software design is moving toward the use of formal methods for specification, design and verification; interface design will eventually follow suit.“

(HEYMANN & DEGANI 2006, S. 25)

4.1 PRINZIPIEN DES VERFAHRENS ZUR ENTWICKLUNG EINES OBERFLÄCHENMODELLS

Im vorigen Kapitel wurde vorgestellt, wie sich ein konkretes User Interface auf seine korrekte Bedienbarkeit hin analysieren lässt. Ausgangspunkt war die Formalisierung des Verhaltens von Systemen und Interfaces durch zustandsbasierte Modelle sowie der Aufgabenanforderung durch Spezifikationsklassen. Auch das Verfahren zur Generierung von Oberflächenmodellen, mit dem sich der nun folgende Abschnitt auseinandersetzt, geht von diesen Gegebenheiten aus: „The machine and the user's operational requirements are given. Now the problem is to generate an interface and associated user information that enables the user to interact with the machine *correctly*. It is further required that the interface and all user information be as simple and succinct as possible“ (HEYMANN & DEGANI 2002 I, S. 4).
Es wird sich herausstellen, dass diese scheinbar auf empirischen Werten und psychologischen Gesichtspunkten basierende Aufgabe des Interface Designs mittels mathematischer Algorithmen lösbar ist, die auf graphentheoretische Überlegungen zurückgehen. Bevor das Verfahren anhand mehrerer Beispiele veranschaulicht werden wird, werden daher die unterliegenden Konzepte untersucht. Das Verfahren selbst wird dann auf abstrakter Ebene sowohl textuell als auch in Form von Pseudocode beschrieben, wobei letzterer aus dem im Rahmen dieser Arbeit entwickelten Software-Tool hervorgeht. Auf die konkrete Umsetzung einzelner Methoden wird anhand der Beispiele genauer eingegangen. Anschließend wird den Fragen nachgegangen, ob das Verfahren den Korrektheitskriterien aus Kapitel 2 standhält, inwiefern Verbesserungen des Verfahrens möglich sind und wo die Grenzen des Verfahrens liegen.

Bereits zu diesem Zeitpunkt lässt sich sagen, dass es nicht zwangsläufig nur *ein* korrektes und minimales Oberflächenmodell zu einem Maschinenmodell und der Aufgabenspezifikation gibt. Es handelt sich bei der Erzeugung des Oberflächenmodells daher nicht um eine Synthese, es wird also nicht versucht, das richtige Modell zu formen. Vielmehr geht es darum, das Maschinenmodell weitestmöglich zu reduzieren ohne die Korrektheitskriterien zu verletzen (VGL. HEYMANN & DEGANI 2002 I, S. 16).

Sei also angenommen, das Maschinenmodell eines Systems und die Spezifikationsklassen sind gegeben. Wie gesagt, braucht der Benutzer zur Aufgabenerfüllung nicht notwendigerweise zwischen einzelnen internen Zuständen unterscheiden zu können, er muss hingegen lediglich stets den aktuellen und nächsten Modus, repräsentiert durch je eine Spezifikationsklasse, kennen. Für ihn erweist es sich deshalb als unerheblich, ob das System in einem Zustand *S* oder einem Zustand *T* ist, falls beide Zustände die gleichen Aktionen ermöglichen und es für das Verfolgen der Spezifikationsklassen ausreicht zu wissen, dass *entweder S oder T* gültig ist. Diese Situation liegt vor, wenn eine beliebige Eingabesequenz gestartet in *S* in die gleiche Klasse führt, die sich gestartet in *T* ergeben würde, und auch die Folgezustände wiederum die gleichen Bedingungen erfüllen (VGL. HEYMANN & DEGANI 2002 I, S. 16). Zwei so beschaffene Zustände sind *spezifikationsäquivalent*, ein Interface kann und sollte sie als ein und denselben Modus anzeigen, da es dem Benutzer nicht helfen würde, sie auseinander halten zu können. Im Gegenteil: Die Information über die interne Differenz ist irrelevant und würde daher eine unnötige Belastung darstellen. In solch einem Fall wird von *kompatiblen* Systemzuständen gesprochen.

Definition 4.1.1: kompatible Zustände / compatible states
(VGL. HEYMANN & DEGANI 2006, S. 12)

Zwei Zustände *S* und *T* eines Systems heißen genau dann **kompatibel**, wenn sie den folgenden drei Bedingungen genügen:

1. *S* und *T* gehören derselben Spezifikationsklasse an.
2. In *S* und *T* können dieselben benutzergesteuerten Ereignisse ausgelöst werden.
3. Jede für *S* und *T* definierte Eingabesequenz führt in Folgezustände *S'* und *T'*, die ebenfalls den Bedingungen 1 und 2 genügen.

Es ist darauf hinzuweisen, dass in HEYMANN & DEGANI 2002 I die zweite Bedingung nicht mit aufgeführt ist, was wohl daraus hervorgeht, dass dort schon in der Definition der Korrektheit die Vermeidung von *augmenting states* ausgelassen wurde (vgl. Kapitel 2.2). Für die Einhaltung dieses Korrektheitskriteriums erweist sich genau die zweite Bedingung nämlich als wichtig.
Wie Definition 4.1.1 erkennen lässt, sind zwei Zustände bei Erfüllung der ersten beiden Bedingungen genau dann kompatibel, wenn für eine beliebige Eingabesequenz gilt, dass entweder das resultierende Zustandspaar *(S', T')* ebenfalls kompatibel ist, oder wenn *S'* und *T'* denselben Zustand bezeichnen. Daher wird die Berechnung der kompatiblen Zustandspaare komplementär angegangen: „Instead of trying to find all state pairs that are compatible, it is computationally more convenient to first find all state-pairs that are *incompatible*" (HEYMANN & DEGANI 2006, S. 13). Entsprechend ergibt sich für Inkompatibilität folgende formale Definition:

Definition 4.1.2: Inkompatible Zustände / Incompatible states
(VGL. HEYMANN & DEGANI 2006, S. 13)

Zwei Zustände *S* und *T* eines Systems heißen genau dann **inkompatibel**, wenn sie mindestens einer der folgenden drei Bedingungen genügen:

1. *S* und *T* gehören nicht derselben Spezifikationsklasse an.
2. *S* und *T* haben nicht die gleichen ausgehenden benutzergesteuerten Transitionen.
3. Es existiert eine für *S* und *T* definierte Eingabesequenz, die in inkompatible Folgezustände *S'* und *T'* führt.

Der erste Schritt des Verfahrens besteht nun darin, alle kompatiblen Zustandsmengen ausfindig zu machen, da sie die Basis für die Reduktion des Maschinenmodells bilden. Dabei ist zu beachten, dass Kompatibilität intransitiv ist, das heißt etwa, aus der Existenz eines kompatiblen Zustandspaars (S_1, S_2) und eines weiteren (S_2, S_3) kann im Allgemeinen keineswegs geschlossen werden, dass auch S_1 und S_3 kompatibel sind. Einerseits folgt hieraus, dass eine Menge von Zuständen nur dann kompatibel ist, wenn ihre Elemente paarweise kompatibel sind. „That is, a *state triple* is compatible if its three constituent state-pairs are compatible, a *state quadruple* is compatible if its four constituent triples are compatible, and so on" (HEYMANN & DEGANI 2006, S. 19). Andererseits zeigt sich, dass die kompatiblen Zustandsmengen innerhalb eines Systems Überlappungen haben können. In Kapitel 4.3 wird solch ein Modell vorgestellt. Gesucht sind nun nicht alle kompatiblen Zustandsmengen, sondern nur diejenigen, deren Umfang nicht erweiterbar ist.

Definition 4.1.3: Maximal kompatible Zustandsmenge / maximal compatible state set
(VGL. HEYMANN & DEGANI 2002 I, S. 18)

> Eine kompatible Menge an Zuständen eines Systems ist **maximal**, wenn sie keine echte Teilmenge einer anderen kompatiblen Zustandsmenge ist.

Ein Zustand des durch den Reduktionsalgorithmus erhaltenen Oberflächenmodells entspricht genau einer maximal kompatiblen Zustandsmenge, also der größtmöglichen Menge an Zuständen, die der Benutzer nicht voneinander zu unterscheiden braucht. Trivialerweise sind Zustände für sich kompatibel, weshalb jeder Zustand mindestens Element einer maximal kompatiblen Menge an Zuständen ist. Da andersherum jeder Systemzustand selbstredend durch einen Zustand des User Interfaces abgedeckt zu sein hat, muss das zu generierende Oberflächenmodell folgerichtig eine *Überdeckung* der Zustände des Maschinenmodells verkörpern.

Definition 4.1.4: Überdeckung / cover
(VGL. HEYMANN & DEGANI 2002 I, S. 17)

> Eine Menge M an kompatiblen Zustandsmengen heißt **Überdeckung** der Zustände eines Systems, wenn jeder Zustand des Systems in mindestens einem Element aus M enthalten ist.

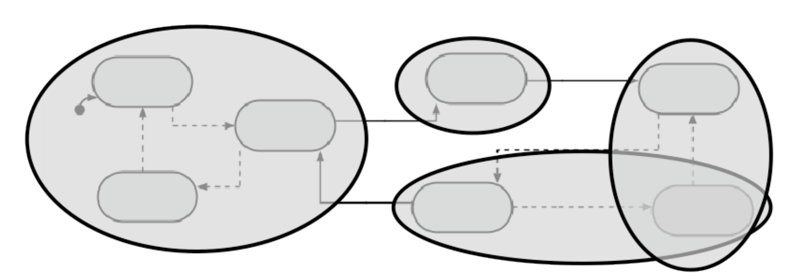


Abbildung 4.1.1: Überdeckung eines Maschinenmodells durch maximal kompatible Zustandsmengen

Abbildung 4.1.1 veranschaulicht die Überdeckung eines Maschinenmodells, zeigt unter Anderem, dass sich zwei maximal kompatible Zustandsmengen überschneiden. Unter der Annahme, dass dem Benutzer alle relevanten Zustandsübergänge offen gelegt werden, gewährleistet das Erfüllen des Überdeckungskriteriums korrekte Bedienbarkeit. Auf die Erarbeitung der Transitionen, die

für das Oberflächenmodell wesentlich sind, wird weiter unten eingegangen. Aufgrund möglicher Überlappungen kann unabhängig davon aber der Fall auftreten, dass für eine Überdeckung nicht alle maximal kompatiblen Zustandsmengen benötigt werden. Da nun weiter ein in dieser Arbeit formuliertes Ziel des User Interface Designs ist, die Anzahl angezeigter Informationen so gering wie möglich halten, bedarf es folglich einer *minimalen* Überdeckung des Maschinenmodells durch maximal kompatible Zustandsmengen:

Definition 4.1.5: Minimale Überdeckung / minimal cover
(VGL. HEYMANN & DEGANI 2002 I, S. 18)

> Eine Überdeckung M der Zustände eines Systems ist **minimal**, wenn keine echte Teilmenge von M eine Überdeckung der Zustände des Systems darstellt.

Maximal kompatible Zustandsmengen und eine minimale Überdeckung des Maschinenmodells erweisen sich also als Essenz des Algorithmus. In Kapitel 4.4 und Kapitel 4.5 wird überprüft werden, ob ein damit gebildetes Oberflächenmodell die Korrektheitskriterien erfüllt und außerdem das Verhalten des unterliegenden Systems weitestmöglich abstrahiert wird. An dieser Stelle soll es genügen, sich der Konzepte bewusst zu sein, da dies erlaubt, den generellen Ablauf des Reduktionsalgorithmus im Gesamten zu betrachten.

4.2 ÜBERBLICK ÜBER DEN REDUKTIONSALGORITHMUS

Ehe die Generierung von Oberflächenmodellen an Beispielen erläutert werden wird, gibt dieses Teilkapitel einen Überblick über das Verfahren. Teilweise wird hier noch nicht die genaue Methodik einzelner Schritte beschrieben und zwar dann, wenn sich dies ohne Beispiel als unpraktikabel erweisen würde. Um einen Bezug zum implementierten Software-Tool zu geben, findet sich im Folgenden die Umsetzung des Verfahrens als Pseudocode auf abstrakter Ebene:

```
// Main method of the reduction algorithm
PROGRAM reduction_algorithm(machineStates, machineTransitions, specClasses)
      VAR maxCompatibles := compute_maximal_compatibles(machineStates,
                  machineTransitions, specClasses)
      VAR userStates := compute_minimal_cover(maxCompatibles, specClasses)
      VAR userTransitions := determine_transitions(userStates,
                  machineStates, machineTransitions)

// Computation of all the maximal compatible state sets
PROCEDURE compute_maximal_compatibles(machineStates, machineTransitions,
            specClasses)
      VAR compatibles := create_initial_compatibles(machineStates)
      VAR mergerTable := compute_initial_resolution(compatibles,
                  specClasses, machineTransitions)
      compatibles += resolve_merger_table_iteratively(mergerTable)
      compute_compatible_state_sets_recursively(compatibles, 3)
      delete_not_maximal_compatibles(compatibles)
      RETURN compatibles
```

```
// Computation of a minimal cover of the machine model by max. compatibles
PROCEDURE compute_minimal_cover(maxCompatibles, specClasses)
      FOR (i:=1 TO #classes) DO
            compsOfClass := get_compatibles_refering_to_class(
                        specClasses[i], maxCompatibles)
            minClassCover := compsOfClass
            FOR (every non-trivial subset m of compsOfClass) DO
                  IF (subset_is_cover_of_class(m, specClasses[i])) THEN
                        IF (#m < #minClassCover) THEN minClassCover := m
            FOR (j:= 1 TO #m) DO userStates += create_user_state(m[j])
      RETURN userStates

// Determine the transitions of the user model by the transitions of the
// machine model
PROCEDURE determine_transitions(userStates, machineStates,
            machineTransitions)
      VAR userTransitions := create_all_transitions(userStates,
                  machineTransitions)
      delete_nondeterministic_innerclass_transitions(userStates,
                  userTransitions)
      delete_automatic_self_loops(userTransitions)
      group_events_that_always_appear_together(userTransitions)
```

Der Reduktionsalgorithmus besteht also, wie die Hauptmethode `reduction_algorithm()` zeigt, aus drei Schritten: Als erstes werden alle maximal kompatiblen Zustandsmengen berechnet (`compute_maximal_compatibles()`), anhand derer dann eine minimale Überdeckung des Maschinenmodells gefunden wird (`compute_minimal_cover()`). Das Resultat dieser beiden Abschnitte sind die Zustände des Oberflächenmodells. Anschließend werden die Transitionen des Modells ermittelt (`determine_transitions()`), die sich aus der Zusammenfassung der Zustände des Systems und den Übergängen zwischen diesen ergeben.

Die Berechnung der maximal kompatiblen Zustandsmengen beginnt mit der Erzeugung der trivial kompatiblen 1-elementigen Mengen. Eine effiziente iterative Methode, die dem Finden aller kompatiblen Zustandspaare dient, basiert auf der Benutzung so genannter *merger tables* (VGL. HEYMANN & DEGANI 2006, S. 13). Eine *merger table* beinhaltet pro Zustandspaar des Maschinenmodells eine Zelle, die in `compute_initial_resolution()` mit den Paaren an Folgezuständen – sofern vorhanden – initialisiert wird, die sich durch Auslösen eines einzelnen Ereignisses erreichen lassen. Zustände, die nicht der gleichen Spezifikationsklasse angehören oder verschiedene Aktionen des Benutzers erlauben, werden schon hier als „inkompatibel" markiert. Danach werden alle weiteren inkompatiblen Zustandspaare mittels iterativen Ersetzens der in den Zellen gespeicherten Paare durch deren mögliche Nachfolgezustandspaare herausgearbeitet, indem die Kriterien aus Definition 4.1.2 angewendet werden. Dieses Resolutionsverfahren, umgesetzt in der Methode `resolve_merger_table_iteratively()`, terminiert, wenn sich die Tabelle nicht mehr verändert, wenn also keine *neuen* Ereignisfolgen mehr abzuhandeln sind. Alle Zustandspaare, die bis dahin nicht als inkompatibel erkannt worden sind, müssen folgerichtig kompatibel sein, denn sie erfüllen die Bedingungen aus Definition 4.1.1.
Es kann nun aber sein, dass sich noch größere kompatible Zustandsmengen finden lassen. Wie angesprochen, folgt für Zustände S_1, S_2 und S_3 etwa aus den kompatiblen Paaren (S_1, S_2),

(S_1, S_3) und *(S_2, S_3)*, dass *(S_1, S_2, S_3)* ein kompatibles Tripel darstellt. Entsprechend sind vier Zustände kompatibel, wenn alle sechs 2-Tupel bzw. alle vier 3-Tupel, die sich bilden lassen, untereinander kompatibel sind. Heymann und Degani beschreiben in den gegebenen Quellen nicht, wie sich die dafür notwendigen Berechnungen vollziehen; vermutlich deshalb, weil dies für eine konkrete Zustandsmenge klar erscheint. Anstatt im allgemeinen Fall aber nun alle kompatiblen Mengen paarweise vergleichen zu müssen, wird innerhalb des Programms ein rekursiver Algorithmus verwendet, der sich Eigenschaften von Mengen und Operationen auf diesen zu Nutze macht und in Kapitel 4.3 näher charakterisiert werden wird. Nach Ausführung der entsprechenden Methode (`compute_compatible_state_sets_recursively()`) sind dann alle kompatiblen Zustandsmengen ermittelt worden. Während der Pseudocode aus Gründen der Übersichtlichkeit an dieser Stelle besagt, dass alle nicht maximalen kompatiblen Mengen jetzt wieder entfernt werden, ist das bei der Umsetzung in der Software bereits passiert: In dem Moment nämlich, in dem etwa wie oben *(S_1, S_2, S_3)* gefunden wurde, kann die Maximalität der drei zugehörigen Paare ausgeschlossen werden.

Der zweite Schritt des Generierungsverfahrens besteht nun darin, *eine* minimale Überdeckung anhand der vorliegenden maximal kompatiblen Zustandsmengen zu bestimmen. „The selection among the various candidate user models cannot, generally, be quantified, and is based on engineering and human-factors considerations“ (HEYMANN & DEGANI 2006, S. 22). Je nach dem, worauf das Design des User Interfaces abzielt, kann gefordert sein, die Anzahl der Zustände oder die der Transitionen zu minimieren. Ebenso kann aber auch die inhaltliche Bedeutung der sich ergebenden Anzeigen das entscheidende Moment sein. Da es hier jedoch darum gehen soll, ein automatisches und allgemein anwendbares Verfahren zu entwickeln und rein prinzipiell kein Kriterium stärker wiegt als die anderen, wurde für das Software-Tool die bestmögliche Reduktion der Menge an Zuständen als oberste Priorität definiert, da sie sicherlich der Entlastung des Benutzers förderlich ist.
Die Berechnung einer minimalen Überdeckung ist ein NP-schweres Problem. Der Beweis dessen soll hier nicht geführt werden, ließe sich aber durch polynomielle Reduktion auf ein bekanntes solches Problem anstellen. Der Code der Methode `compute_minimal_cover()` zeigt, dass pro Spezifikationsklasse alle Kombinationen zugehöriger maximal kompatibler Zustandsmengen (`compsOfClass`) getestet werden, um eine minimale Überdeckung der Zustände der Klasse (`minClassCover`) zu finden. Anschließend werden aus den ausgewählten Zustandsmengen direkt Zustände des Oberflächenmodells erzeugt (`create_user_state()`). Selbstredend wird mit diesem *brute-force*-Ansatz die Minimierung anzuzeigender Zustände garantiert.

Auf Basis der durch die Überdeckung gegebenen Zustände werden schließlich die Transitionen des Oberflächenmodells festgesetzt. Generell darf verständlicherweise kein Ereignis, welches in einem Wechsel des Zustands des Oberflächenmodells resultiert, verschwiegen werden. Aus diesem Grund werden in `create_all_transitions()` vorerst alle Übergänge, die zwischen den in diesen Zuständen zusammengefassten Systemzuständen hin- und herschalten, generiert: „for each user model mode and each event label that emanates from it, we determine the set of all constituent machine model target states“ (HEYMANN & DEGANI 2006, S. 22). Angenommen also, ein Zustand S des Interfaces beschreibt die maximal kompatible Zustandsmenge $\{S_1, S_2\}$. Intern führe ein Ereignis β von S_1 zu T_1, von S_2 aus hingegen zu T_2 (daraus folgt nebenbei, dass

T_1 und T_2 kompatibel sein müssen). Dann werden mit β gelabelte Transitionen zwischen S und allen Interface-Zuständen, die sowohl T_1 als auch T_2 beinhalten, erstellt.
Offensichtlich können sich nicht-deterministische Situationen ergeben, wenn mehrere Zustände aus internen Folgezuständen (hier: aus T_1 und T_2) hervorgehen. Dieser Nicht-Determinismus kann aber nie in *error states* münden, weil er nur innerhalb von Spezifikationsklassen auftritt (VGL. HEYMANN & DEGANI 2006, S. 23). Er lässt sich beseitigen, indem redundante Transitionen einfach entfernt werden (`delete_nondeterministic_innerclass_transitions()`). Dabei werden bevorzugt solche Übergänge bewahrt, die wieder direkt zum Ausgangszustand zurückführen. Denn `delete_automatic_self_loops()` löscht im nächsten Schritt entweder alle automatischen *self-loops* oder nur diejenigen, deren zugehöriges Ereignis ausschließlich als *self-loop* vorkommt. Die Benutzeroberfläche der Software hat dafür eine Einstellungsmöglichkeit: „it is up to the design team to decide, based on operational and situation awareness consideration, whether they want to provide information about these internal gear shifts [...]" (HEYMANN & DEGANI 2006, S. 19). Hat auch diese Reduktion stattgefunden, so können zuletzt noch Ereignisse zusammengefasst werden, die stets zusammen an Transitionen erscheinen. In diesem Fall nämlich erweist sich eine Unterscheidung zweier solcher Ereignisse für den Benutzer als irrelevant.
Aus den berechneten Transitionen und den zuvor konstruierten Zuständen lässt sich im Anschluss das User Interface ableiten. Hierfür muss noch entschieden werden, welche Informationen direkt darzustellen sind und welche einer Bedienungsanleitung oder anderen externen Quellen entnommen werden sollen.

Der Reduktionsalgorithmus terminiert also mit dem bisher als korrekt und minimal vermutetem Modell der Oberfläche. Er wurde in diesem Abschnitt vollständig aber abstrakt beschrieben. Zur Veranschaulichung finden sich daher im nächsten Teilkapitel zwei Beispiele, die die wesentlichen Eigenschaften des Verfahrens illustrieren. Zuerst geht es darum, das korrekte und minimale User Interface des Heiz- und Kühlsystems zu ermitteln. Danach werden ausstehende Aspekte des Reduktionsalgorithmus anhand eines abstrakten Systems vorgestellt. Ob die resultierenden Oberflächenmodelle wirklich die Korrektheitskriterien erfüllen und auch nicht weiter reduziert werden können, ist anschließend Thema von Kapitel 4.4. Wie bereits in Aussicht gestellt wurde, wird sich dort untere anderem eine Schwachstelle in der Definition der Spezifikationsklassen zeigen, der sich mithilfe einer zusätzlichen Bedingung entgegenwirken lässt.

4.3 KONKRETE ANWENDUNG DES ALGORITHMUS ZUR GENERIERUNG VON OBERFLÄCHENMODELLEN

In Kapitel 3 wurde aufgezeigt, dass die intuitive Erzeugung eines korrekten und minimalen Oberflächenmodells schwer durchführbar ist. Das bereits dort eingeführte Maschinenmodell des halbautomatischen Heiz- und Kühlsystems (vgl. Abbildung 3.3.1) wird nun dazu dienen, den soeben behandelten Ablauf des Reduktionsalgorithmus zu demonstrieren.
Die Berechnung der Zustände des User Interfaces startet mit den trivialerweise kompatiblen Zustandsmengen, welche durch die neun internen Zustände gegeben sind. Anhand dieser wird

die initiale Resolution der *merger table* ermittelt. Sei zuvor jedoch das Vorgehen der entwickelten Software betrachtet, das der folgende Pseudocode zusammenfasst. Zur Vereinfachung sei angenommen, dass die Zustände des Maschinenmodells von oben nach unten und links nach rechts durchnummeriert sind.

```
// Compute the initial resolution of the merger table
PROCEDURE compute_initial_resolution(states, classes, transitions)
      VAR mergerTable := new MergerTable[#states][#states]
      FOR (i:=1 TO #states) DO
            FOR (j:=0 TO i-1) DO
                  IF (classes[states[i]] != classes[states[j]]) THEN
                        mergerTable[i][j] := "Incompatible"
                  ELSE IF (transitions[states[i],"user-triggered"] !=
                              transitions[states[j],"user-triggered"]) THEN
                         mergerTable[i][j] := "Incompatible"
                  ELSE mergerTable[i][j] :=
                              compute_initial_related_state_pairs(states[i],
                              states[j],transitions)
      RETURN mergerTable
```

Die **`IF`**-Bedingungen regeln also, dass eine Zelle als "inkompatibel" markiert wird, wenn die zwei zugehörigen Zustände nicht der gleichen Spezifikationsklasse angehören oder nicht dieselben ausgehenden benutzergesteuerten Transitionen aufweisen. Für alle verbleibenden Zustandspaare werden die Zellen mit den Zustandspaaren gefüllt, die sich durch Auslösen eines gemeinsamen Ereignisses erreichen lassen (`compute_initial_related_state_pairs()`). Die initiale *merger table* besteht aus 36 (= 9*8/2) Zellen und ist in Abbildung 4.3.1 dargestellt.

	Cool-A	Cool-B	Cool-C	Warm-A	Warm-B	Warm-C	Hot-A	Hot-B
Cool-B	[Warm-B Warm-C] [Cool-B Cool-C]							
Cool-C	[Warm-B Hot-C]	[Warm-C Hot-C] [Cool-A Cool-B]						
Warm-A	Incompatible	Incompatible	Incompatible					
Warm-B	Incompatible	Incompatible	Incompatible					
Warm-C	Incompatible	Incompatible	Incompatible	[Hot-A Hot-C] [Cool-A Cool-B]	[Hot-A Hot-C] [Cool-A Cool-B]			
Hot-A	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		
Hot-B	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	[Warm-A Warm-B] [Hot-B Hot-C]	
Hot-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	[Warm-A Warm-B]	[Hot-A Hot-B]

Abbildung 4.3.1: Initiale Resolution der merger table des Heiz- und Kühlsystems

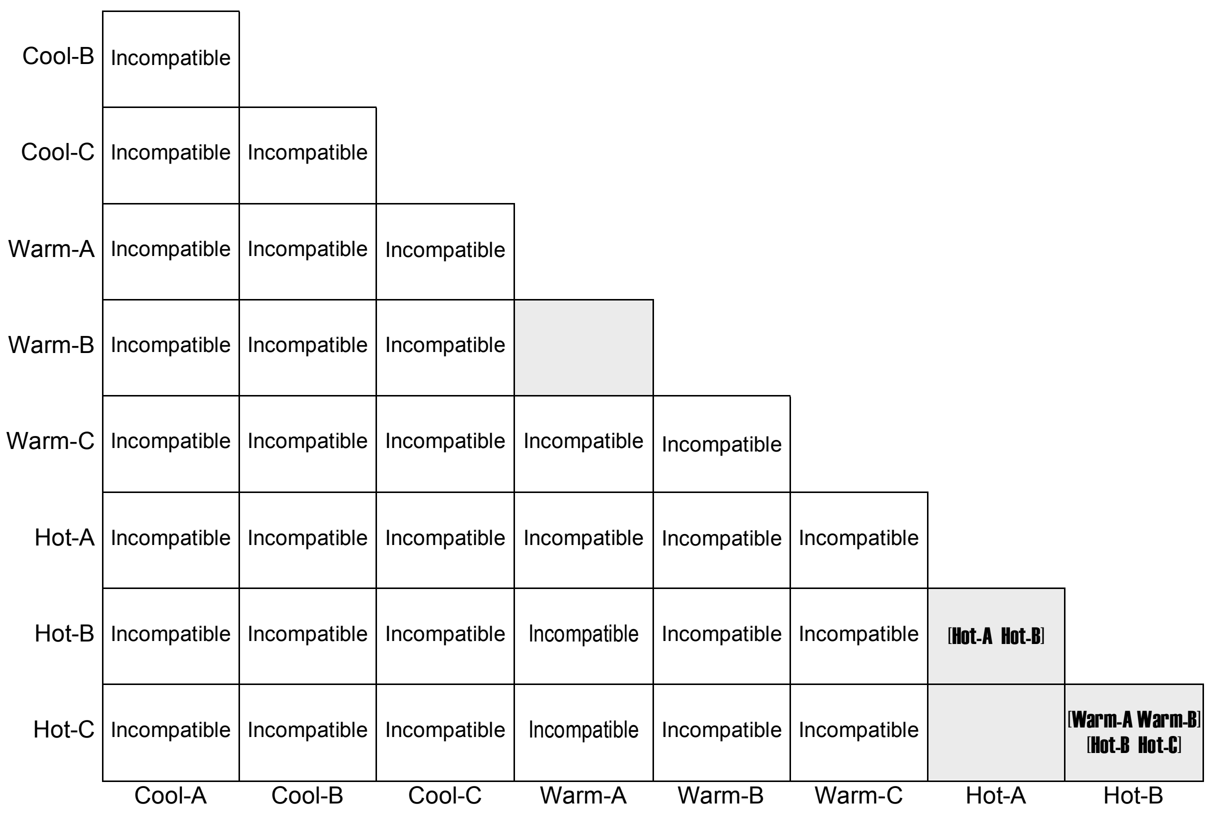
Dabei sind Zellen, deren zugehörige Zustände noch nicht als inkompatibel identifiziert wurden, grau hinterlegt. Folgezustandspaare sind außerdem zur Abgrenzung gegenüber kompatiblen Zuständen in einer anderen Schriftart dargestellt. Beginnend bei der obersten Zelle in Abbildung 4.3.1, die die Zustände Cool-A und Cool-B repräsentiert, finden sich zwei Folgezustandspaare; mit Blick auf das Maschinenmodell zeigt sich, dass die Transition *heat up* zu Warm-B bzw. Warm-C führt. Daher wird [Warm-B Warm-C] in die Zelle geschrieben. Das automatische Ereignis *cool+* schaltet hingegen nach Cool-B bzw. Cool-C. Bei der Zelle darunter, also der von Cool-A und Cool-C, fällt auf, dass kein gemeinsames internes Ereignis existiert, ein manuelles Auslösen von *heat up* bewirkt aber einen Wechsel zu [Warm-B Hot-C].
So wird der Inhalt der Tabelle Zelle für Zelle ermittelt. Bereits hier kann die Kompatibilität eines großen Anteils aller Zustände ausgeschlossen werden (vgl. Abbildung 4.3.1), so etwa bei Cool-A und Warm-B. Dies lässt sich auf verschiedene Spezifikationsklassen zurückführen, aber auch dem Vergleich der benutzergesteuerten Transitionen hätte keins der inkompatiblen Zustandspaare standgehalten. Die zu Warm-A und Warm-B gehörige Zelle ist nun nicht leer, weil die beiden Zustände inkompatibel sind. Es gibt lediglich kein Ereignis, das von ihnen aus in ein anderes Zustandspaar führt. Die gemeinsamen Transitionen *heat up* und *cool down* führen nämlich jeweils in den gleichen Zustand, können somit für keine fehlerhafte Situation verantwortlich sein. Schon jetzt lässt sich konstatieren, dass sich diese Zustände als kompatibel herausstellen werden.

Nach Berechnung der initialen Resolution wird die Tabelle iterativ resolviert. Eine Verwendung des Maschinenmodells ist von hier an nicht mehr notwendig, der aktuelle Inhalt der *merger table* ist vollkommen ausreichend (VGL. HEYMANN & DEGANI 2006, S. 15). Als „inkompatibel" markierte Zellen werden nicht weiter verändert. Jede andere Zelle ist wie folgt zu substituieren: Falls sie einen Verweis auf eine als „inkompatibel" markierte Zelle beinhaltet, wird sie ebenfalls als solche gekennzeichnet, so etwa befindet sich in der Zelle von Cool-B und Cool-C das inkompatible Zustandspaar [Warm-C Hot-C]. Ansonsten wird jedes Zustandspaar durch alle Paare ersetzt, die in der Originalzelle des Zustandspaars notiert sind. Die Zelle von Hot-B und Hot-C zum Beispiel enthält das Paar [Hot-A Hot-B]. Die Originalzelle dieser Zustände enthält wiederum [Warm-A Warm-B] sowie [Hot-B Hot-C], so dass [Hot-A Hot-B] mit diesen überschrieben wird. Dieses Vorgehen liefert die Tabellenbelegung aus Abbildung 4.3.2 auf der folgenden Seite und entspricht dem folgenden Pseudocode (die Bedeutung der Variable `cellChanged` wird weiter unten aufgegriffen).

```
// Resolve the merger table as long as cells are changing
PROCEDURE resolve_merger_table_iteratively(mergerTable)
      VAR cellChanged := TRUE
      WHILE (cellChanged = TRUE)
            cellChanged := FALSE
            FOR (i:=1 TO #states) DO
                  FOR (j:=0 TO i-1) DO
                        IF (mergerTable[i][j] != "Incompatible") THEN
                              VAR oldCell := mergerTable[i][j]
                              substitute_cell(mergerTable, i, j)
                              IF (oldCell != mergerTable[i][j]) THEN
                                    cellChanged := TRUE
      VAR newCompatibles := create_new_compatibles(mergerTable)
      RETURN newCompatibles
```

	Cool-A	Cool-B	Cool-C	Warm-A	Warm-B	Warm-C	Hot-A	Hot-B
Cool-B	[Hot-A Hot-C] [Cool-A Cool-B] [Warm-C Hot-C] [Cool-A Cool-B]							
Cool-C	Incompatible	Incompatible						
Warm-A	Incompatible	Incompatible	Incompatible					
Warm-B	Incompatible	Incompatible	Incompatible					
Warm-C	Incompatible	Incompatible	Incompatible	[Warm-A Warm-B] [Warm-B Warm-C] [Cool-B Cool-C]	[Warm-A Warm-B] [Warm-B Warm-C] [Cool-B Cool-C]			
Hot-A	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		
Hot-B	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	[Hot-A Hot-B]	
Hot-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		[Warm-A Warm-B] [Hot-B Hot-C]

Abbildung 4.3.2: Erste Iteration des Resolutionsprozesses der merger table



	Cool-A	Cool-B	Cool-C	Warm-A	Warm-B	Warm-C	Hot-A	Hot-B
Cool-B	Incompatible							
Cool-C	Incompatible	Incompatible						
Warm-A	Incompatible	Incompatible	Incompatible					
Warm-B	Incompatible	Incompatible	Incompatible					
Warm-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible			
Hot-A	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		
Hot-B	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	[Hot-A Hot-B]	
Hot-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		[Warm-A Warm-B] [Hot-B Hot-C]

Abbildung 4.3.3: Zweite Iteration des Resolutionsprozesses der merger table

In der nächsten Iteration werden drei weitere inkompatible Zustandsmengen entdeckt (siehe Abbildung 4.3.3). So erweisen sich beispielsweise Warm-A und Warm-B beide inkompatibel mit Warm-C, denn in den zugehörigen Zellen taucht jeweils [Cool-B Cool-C] auf, das sich ja zuvor schon

als inkompatibel herausgestellt hatte. Außerdem zeigt sich bereits hier, dass die Zellen der Paare aus **HOT** gleich bleiben: Sowohl die Zelle von Hot-A und Hot-B als auch die von Hot-B und Hot-C verweisen auf sich selbst. Da darüber hinaus [**Warm-A Warm-B**] leer ist, gibt es keine Änderung des Inhalts der Zellen.

Ein weiterer Schritt des Resolutionsprozesses identifiziert keine neuen inkompatiblen Paare. Ebenso ist der Inhalt der Tabelle identisch zu dem in Abbildung 4.3.3. Innerhalb der Umsetzung des Algorithmus gilt also „`cellChanged = FALSE`", was zum Abbruch der Schleife führt. Die Resolution terminiert aus diesem Grund und die Zellen der Zustände, die nicht „Incompatible" in sich tragen, werden als „Compatible" markiert, wie es in Abbildung 4.3.4 dargestellt ist.

Cool-B	Incompatible							
Cool-C	Incompatible	Incompatible						
Warm-A	Incompatible	Incompatible	Incompatible					
Warm-B	Incompatible	Incompatible	Incompatible	Compatible				
Warm-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible			
Hot-A	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible		
Hot-B	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Compatible	
Hot-C	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Incompatible	Compatible	Compatible
	Cool-A	Cool-B	Cool-C	Warm-A	Warm-B	Warm-C	Hot-A	Hot-B

Abbildung 4.3.4: Endgültige Resolution der merger table

Das heißt, die Resolutionsmethode hat vier kompatible Zustandspaare herausgearbeitet, nämlich {Warm-A, Warm-B}, {Hot-A, Hot-B}, {Hot-A, Hot-C} und {Hot-B, Hot-C}. Alle anderen Zustände dürfen zur Gewährleistung der Beobachtbarkeit des Systems niemals zusammengefasst werden, insbesondere darf entgegen den in Kapitel 3.3 vermuteten Oberflächenmodellen keiner der Zustände aus **COOL** verheimlicht werden. Die Zustandspaare werden am Ende der Methode `resolve_merger_table_iteratively()` erzeugt und anschließend zurückgegeben. Doch stellen sie noch nicht zwangsläufig die maximal kompatiblen Zustandsmengen dar, welche gesucht sind, um das Maschinenmodell bestmöglich zu reduzieren. Offensichtlich sind Hot-A, Hot-B und Hot-C paarweise kompatibel, weswegen es sich bei {Hot-A, Hot-B, Hot-C} um eine kompatible Zustandsmenge der Größe 3 handelt. {Warm-A, Warm-B} ist hingegen maximal, denn es existiert kein weiteres Paar derselben Spezifikationsklasse.

Das intuitive Erkennen kompatibler Zustandsmengen ist für kleine Beispiele leicht, wird ab einer gewissen Größe der Zustandsmengen jedoch unübersichtlich. Wie geht also ein Algorithmus im

Allgemeinen vor? Das rekursive Verfahren des entwickelten Software-Tools zur Berechnung der kompatiblen Zustandsmengen ist im folgenden Pseudocode skizziert. Die Methode bricht ab, wenn in einem Durchlauf keine neuen kompatiblen Mengen mehr gefunden worden sind („foundNewCompatible = FALSE").

```
// Find all compatible state sets with cardinality n > 2 recursively
PROCEDURE compute_compatible_state_sets_recursively(compatibles, n)
      VAR foundNewCompatible := FALSE
      VAR comps := get_compatibles_of_size(n-1)
      FOR (i:=1 TO #comps-1) DO
            FOR (j:=0 TO i-1) DO
                  VAR union := get_union(comps[i], comps[j])
                  IF (#union = n) THEN
                        VAR newCompatible := TRUE
                        VAR intersection := get_intersection(comps[i],
                              comps[j])
                        FOR (k:=1 TO #intersection) DO
                              VAR set := union - intersection[k]
                              IF NOT (contains(comps, set)) THEN
                                    newCompatible := FALSE
                        IF (newCompatible = TRUE) THEN
                              compatibles += create_new_compatible(union)
                              foundNewCompatible := TRUE
      IF (foundNewCompatible = TRUE) THEN
            compute_compatible_state_sets_recursively(compatibles, n+1)
```

Anhand der Zustände aus **HOT** sei diese Methode zur Berechnung von Mengen der Größe $n = 3$ erklärt: Die Vereinigung zweier kompatibler Zustandspaare, seien dies etwa {Hot-A, Hot-B} und {Hot-B, Hot-C}, ist U= {Hot-A, Hot-B, Hot-C}, welche in „get_union(comps[i], comps[j])" konstruiert wird. U enthält einen Zustand mehr als die Eingabemengen, ist also ein potentieller Kandidat für eine neue kompatible Menge („#union = n"). Daher wird die Schnittmenge I = {Hot-B} der Zustandspaare gebildet.

Existieren nun alle kompatiblen Zustandsmengen der Größe $n\text{-}1 = 2$, die nötig für paarweise Kompatibilität sind, so stellt U eine neue kompatible Zustandsmenge dar. Sie werden schrittweise in „VAR set := union - intersection[k]" erzeugt. Es wird also jeweils die Differenz von U und einem Element aus I betrachtet, denn gesucht sind genau alle die möglichen Teilmengen M von U, für die $\#M = \#U\text{-}1$ gilt. Im vorliegenden Fall ist das neben den Ausgangspaaren nur {Hot-A, Hot-C}. Da auch {Hot-A, Hot-C} zu den kompatiblen Mengen der Größe $n\text{-}1 = 2$ gehört („contains(comps, set)"), muss {Hot-A, Hot-B, Hot-C} folglich ebenfalls eine kompatible Zustandsmenge der Größe $n = 3$ sein.

{Hot-A, Hot-B}, {Hot-A, Hot-C} und {Hot-B, Hot-C} sind somit im Umkehrschluss nicht maximal. Eine kompatible Menge mit Kardinalität 4 ist beim Heiz- und Kühlsystem ausgeschlossen, da ohnehin keine Spezifikationsklasse mehr als drei Zustände umfasst. Der erste Schritt des Reduktionsalgorithmus liefert also die sechs maximal kompatiblen Zustandsmengen {Cool-A}, {Cool-B}, {Cool-C}, {Warm-A, Warm-B}, {Warm-C} und {Hot-A, Hot-B, Hot-C}.

Die Reduktion hat eine Menge an maximal kompatiblen Zustandsmengen berechnet, in deren Elementen es keine Überlappungen gibt. Die minimale Überdeckung des Maschinenmodells enthält deshalb all diese Zustandsmengen. Sie bilden die Basis für die Erstellung der Zustände des Oberflächenmodells. Zur Unterscheidung werden Cool-A, Cool-B und Cool-C in COOL-1, COOL-2 und COOL-3 umbenannt. Der aus Warm-A und Warm-B resultierende Zustand erhält das Label WARM-1, aus Warm-C wird WARM-2. Da alle Zustände aus **HOT** kompatibel sind, bekommt der aus ihnen hervorgehende Zustand schließlich die Benennung HOT.
Aus der Eindeutigkeit der Zustandsüberdeckung folgt weiterhin, dass im Grunde keine der Transitionen, die in `create_all_transitions()` generiert werden, wieder zu entfernen ist. Eine Analyse der Methode `determine_transitions()` gibt hier nicht viel her und wird auf das folgende Beispiel verschoben. Abbildung 4.3.5 zeigt die Ausgabe des Reduktionsalgorithmus: das korrekte und minimale Oberflächenmodell des Heiz- und Kühlsystems. Nur die entstandenen *self-loops* an WARM-1 und HOT können bei Bedarf internalisiert werden. Diese automatischen Ereignisse bewirken keine Zustandsänderung im Oberflächenmodell, ihre Anzeige würde lediglich der Mitteilung über interne Vorgänge dienen (VGL. HEYMANN & DEGANI 2006, S. 19). Wie erwähnt, lässt sich diese Wahl im Software-Tool einstellen.

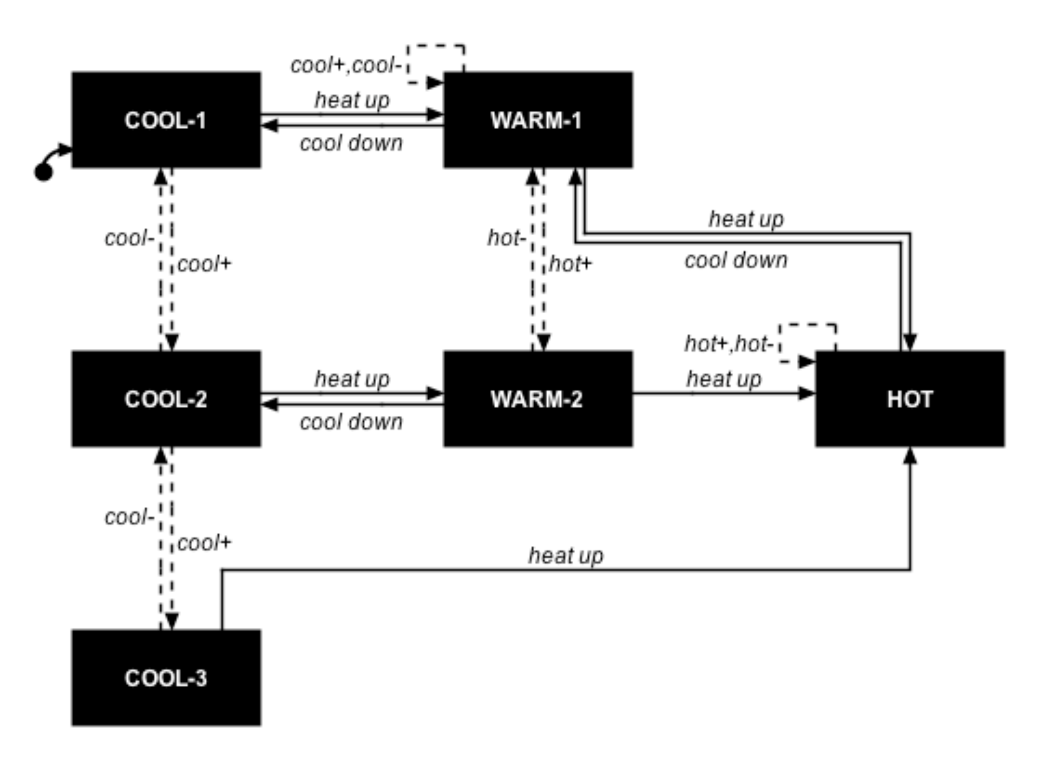


Abbildung 4.3.5: Korrektes und minimales Oberflächenmodell des Heiz- und Kühlsystems

Was nun verbleibt, ist die konkrete Überprüfung der Korrektheit des Oberflächenmodells (die Minimalität wird im Kapitel 4.5 anhand des Beweises der Korrektheit des Verfahrens gezeigt). Dazu wird die in Kapitel 3.3 vorgestellte Methode auf das Oberflächenmodell angewendet. Die Herleitung des sich ergebenden zusammengesetzten Modells sei hier vernachlässigt, das Modell findet sich in Abbildung 4.3.6. Es weist keine *error states, restricting states* und *augmenting states* auf und erfüllt damit die Korrektheitskriterien aus Definition 2.2.1.

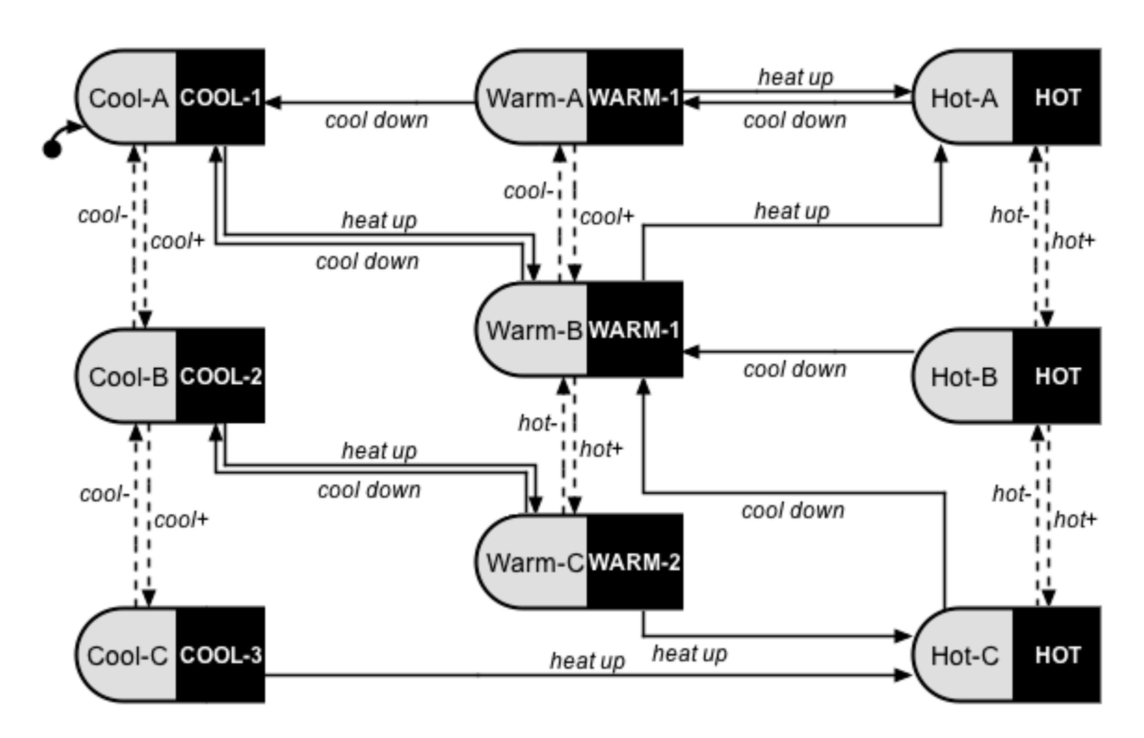


Abbildung 4.3.6: Zusammengesetztes Modell des Heiz- und Kühlsystems

Bei der Generierung des Oberflächenmodells des Heiz- und Kühlsystems sind nicht alle Aspekte des Reduktionsalgorithmus zum Tragen gekommen. Daher sei hier noch ein weiteres Beispiel kurz betrachtet, das insbesondere mehr Aufschluss über die Überlappungen von kompatiblen Zustandsmengen und das Festsetzen von Transitionen geben wird. Abbildung 4.3.7 zeigt das Maschinenmodell eines abstrakten Systems, dessen inhaltliche Bedeutung hier nicht weiter von Interesse ist. Der Benutzer soll auch dieses Mal zu jedem Zeitpunkt über die aktuelle und folgende Spezifikationsklasse bescheid wissen.

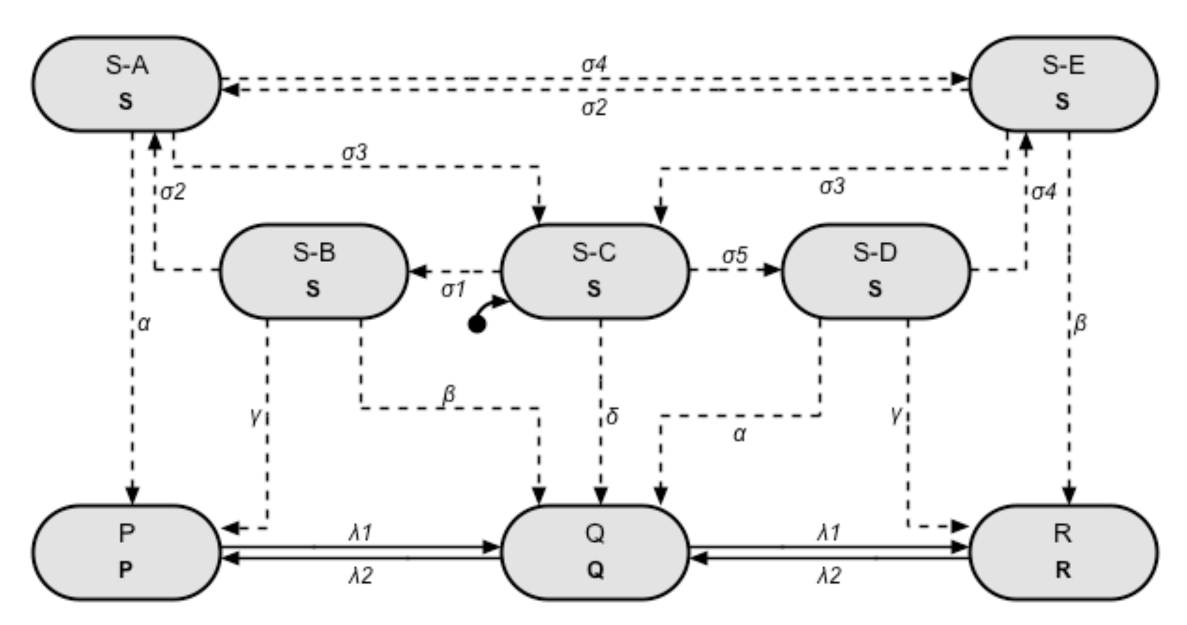


Abbildung 4.3.7: Maschinenmodell eines abstrakten Systems inklusive Spezifikationsklassen

Die Ereignisse ***σ1, σ2, σ3, σ4*** und ***σ5*** können zu Beginn automatisch innerhalb der Spezifikationsklasse **S** Übergänge triggern. Ein Auftreten von *α, β, γ* oder *δ* schaltet dann in

einen der eindeutig zu unterscheidenden Zustände P, Q und R, in denen der Benutzer Aktionen ausführen kann.
Dieses System sei nun die Eingabe des Reduktionsalgorithmus. Im ersten Schritt werden die maximal kompatiblen Zustandsmengen {S-A, S-B, S-C}, {S-C, S-D, S-E} und {S-A, S-C, S-E} sowie {P}, {Q} und {R} berechnet. Offensichtlich finden sich hier, anders als beim Heiz- und Kühlsystem, Überschneidungen zwischen den kompatiblen Mengen der Klasse **S**. Es zeigt sich, dass {S-A, S-C, S-E} für die minimale Überdeckung überflüssig ist, alle anderen werden hingegen benötigt. In Abbildung 4.3.8 ist die minimale Überdeckung des Maschinenmodells dargestellt. Bereits hinzugefügt wurden die vorerst erzeugten Transitionen. Da der Startzustand des Systems (S-C) sowohl in {S-A, S-B, S-C} als auch {S-C, S-D, S-E} enthalten ist, kann aus diesen beiden der entsprechende Startzustand des User Interfaces frei gewählt werden und so findet sich die zugehörige Markierung in der Abbildung bei {S-C, S-D, S-E}.

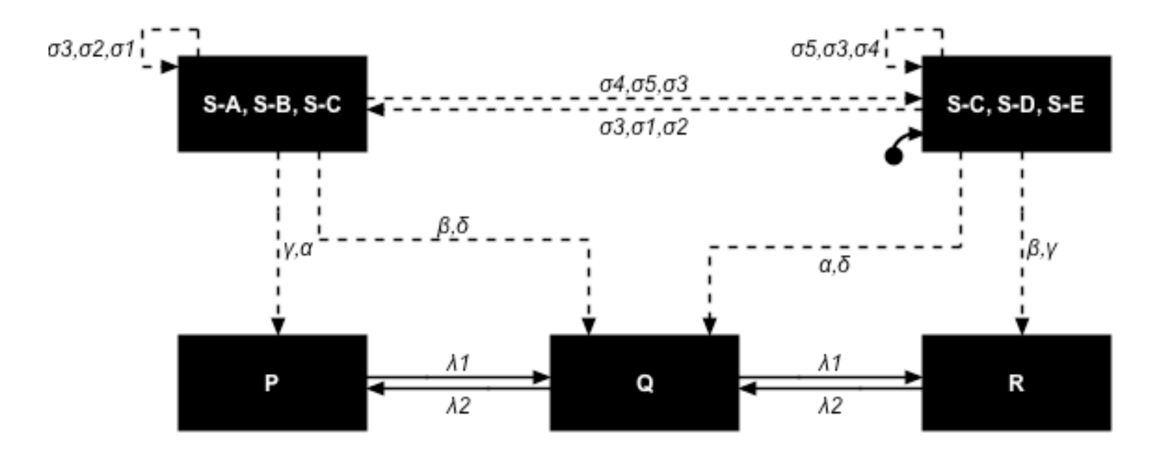


Abbildung 4.3.8: Reduziertes Maschinenmodell des abstrakten Systems nach Erstellung aller Transitionen

Bei Betrachtung der Übergänge der oberen Zustände des Interfaces, die jeweils drei Zustände des Systems repräsentieren, fällt auf, dass sie beide nicht-deterministische ausgehende Transitionen namens *σ3* aufweisen. Ein Ereignis wie *σ3* löst jedoch nur Zustandswechsel innerhalb der Spezifikationsklasse aus (vgl. Kapitel 4.2) und so lässt sich der Nicht-Determinismus aufheben, indem redundante Transitionen gelöscht werden. In diesem Fall geschehe das bei den Übergängen, die zwischen {S-A, S-B, S-C} und {S-C, S-D, S-E} verlaufen. Als Resultat ergibt sich, dass *σ3* nur noch als *self-loop* auftritt und da derlei interne Ereignisse keinen Einfluss auf den angezeigten Zustand des User Interfaces haben, können auch die weiteren mit *σ3* gelabelten Transitionen entfernt werden.
Ferner ist ersichtlich, dass *σ1* und *σ2* ebenso wie *σ4* und *σ5* ausschließlich zusammen an den Übergängen auftreten. Genau wenn diese Situation vorliegt, können zwei solche Ereignisse gruppiert werden, so dass sie im Oberflächenmodell als ein und dasselbe Ereignis erscheinen. Daher werden hier *σ1* und *σ2* zu *σ-a* abstrahiert, *σ4* und *σ5* zu *σ-b*. Da das Software-Tool der sinnvollen Vergabe von Labels kaum gerecht werden kann, würden dort beispielsweise *σ4* und *σ5* einfach zu *„σ4/σ5"* zusammengezogen, was der Benutzer im Nachhinein aber noch ändern könnte. Unabhängig davon werden die Bezeichnungen der zu P, Q und R führenden Transitionen hingegen beibehalten, denn die zugehörigen Ereignisse treten in verschiedener Kombination auf und so kann eine Unterscheidung selbiger im Allgemeinen wichtig sein.

Zuletzt seien noch die Zustände {S-A, S-B, S-C} in S-1 und {S-C, S-D, S-E} in S-2 umbenannt. Es entsteht das in Abbildung 4.3.9 aufgeführte korrekte und minimale Oberflächenmodell des Systems.

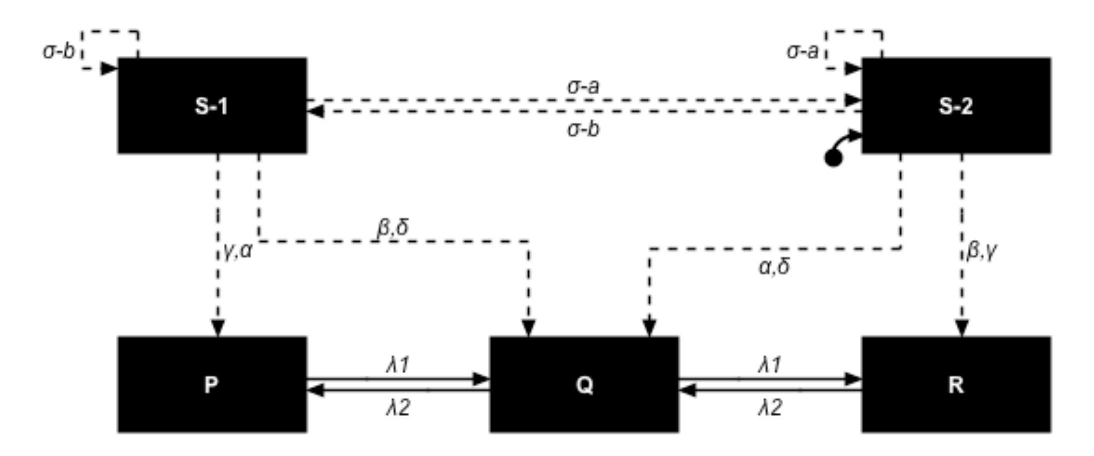


Abbildung 4.3.9: Korrektes und minimales Oberflächenmodell des abstrakten Systems

Damit sind alle grundlegenden Vorgehensweisen des Reduktionsalgorithmus besprochen. Auf die explizite Angabe der in `determine_transitions()` enthaltenen Methoden wird verzichtet, da es sich bei den dort angestellten Berechnungen im Wesentlichen um Überprüfungen der im Maschinenmodell bestehenden Transitionen handelt. Kapitel 5 wird sich noch den technischen Hintergründen des Software-Tools widmen. Ebenso wird die Oberfläche des Programms samt ihren Funktionalitäten dort beschrieben und deren Benutzung exemplarisch durchgeführt.
Im Fokus soll jetzt erst einmal die Frage stehen, ob der Reduktionsalgorithmus die Korrektheitskriterien einhält. Es wird sich zeigen, dass weder *restricting states* noch *augmenting states* im allgemeinen Fall ausgeschlossen sind. Verbesserungen des Verfahrens werden vorgestellt, anhand derer sich danach die Korrektheit des Algorithmus beweisen lässt.

4.4 FEHLERBEHEBUNG UND DAFÜR NÖTIGE ÄNDERUNGEN AM REDUKTIONSALGORITHMUS

Schon in Kapitel 3.2 wurde darauf hingewiesen, dass im Rahmen dieser Arbeit ein Fehler im Verfahren zur Generierung von User Interfaces entdeckt wurde. Unter der gegebenen Definition von Spezifikationsklassen (vgl. Definition 3.2.1) vermeidet das berechnete Oberflächenmodell nicht in allen Fällen das Auftreten von *augmenting states*. Ein Gegenbeispiel wird diese Behauptung nun zuerst belegen. Anschließend wird eine mögliche Behebung des Problems durch Einführen einer zusätzlichen Bedingung vorgestellt, die das Verfahren jedoch einschränkt. Desweiteren lassen sich Fälle konstruieren, in denen der Reduktionsalgorithmus von Heymann und Degani Oberflächenmodelle erzeugt, die *restricting states* enthalten. Auch für dieses Problem wird ein Ausweg aufgezeigt werden, der eine Änderung des Festsetzens der im Oberflächenmodell anzuzeigenden Transitionen vornimmt. Dieser Ausweg legt gleichzeitig eine Erweiterung der Korrektheitskriterien um die Vermeidung noch zu definierender *concealing states* nahe, wird leider aber auch in gewisser Hinsicht Probleme offen lassen.

Satz 4.4.1:

Unter der Definition von Spezifikationsklassen (Definition 3.2.1) gewährleistet der Reduktionsalgorithmus nicht die Vermeidung von augmenting states.

Beweis von Satz 4.4.1 über Widerspruch: Angenommen, der Algorithmus vermeidet *augmenting states*. Dann muss er deren Existenz in allen berechneten Oberflächenmodellen generell ausschließen. Es werde nun folgendes Beispiel betrachtet:
Abbildung 4.4.1 zeigt das abstrakte Maschinenmodell eines Systems. Die Aufgabenspezifikation besage, dass der Benutzer jederzeit weiß, in welcher der Spezifikationsklassen **C-1** und **C-2** sich das System befindet und welche es als nächstes erreichen wird. Das benutzergesteuerte Ereignis β schaltet vom Startzustand S-1 in den Zustand S-2. In S-2 bleibt ein weiteres Auftreten von β ergebnislos. Das interne Ereignis τ kann jedoch einen Zustandswechsel nach T-1 bewirken.
Das Beispiel ist keineswegs unrealistisch. Wie bereits zu Beginn von Kapitel 3 gesagt, geben viele Fußgängerampeln nach Drücken des Knopfes kein Feedback. Die Grünphase für den Fußgänger kommt hingegen einfach scheinbar unbestimmte Zeit später.

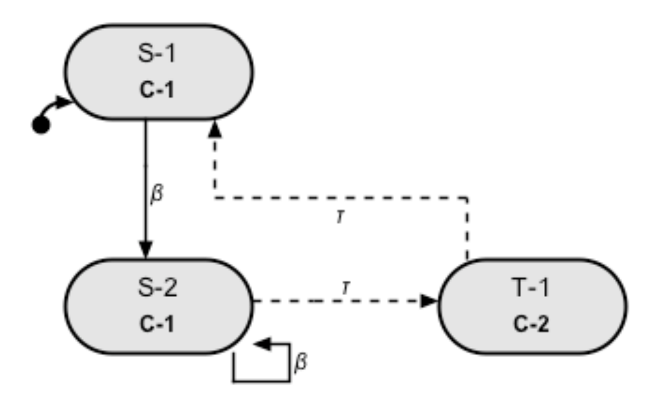


Abbildung 4.4.1: Maschinenmodell eines abstrakten Systems inklusive zwei Spezifikationsklassen

Welches Oberflächenmodell ergibt sich also nach Anwendung des Reduktionsalgorithmus? In Abbildung 4.4.2 findet sich die initiale Resolution der *merger table;* T-1 ist inkompatibel mit beiden anderen Zuständen, denn er gehört nicht der gleichen Spezifikationsklasse an. Weiter gibt es kein gemeinsames Ereignis von S-1 und S-2, das in einem Zustandspaar resultiert. Daher bleibt die Zelle, welche diese Zustände repräsentiert, leer.
Schon nach einer einzigen Iteration des Resolutionsprozesses verändert sich die Tabelle nicht mehr. Die Zelle von S-1 und S-2 wird folglich als „kompatibel" markiert (vgl. Abbildung 4.4.3).

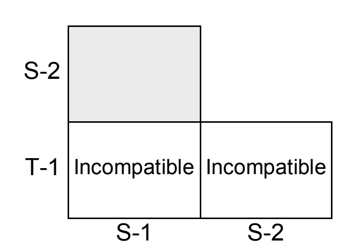


Abbildung 4.4.2: Initiale Resolution der merger table

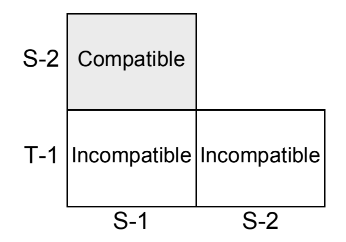


Abbildung 4.4.3: Endgültige Resolution der merger table

Ein kompatibles Zustandstripel ist offensichtlich unmöglich, die Resolution terminiert mit den beiden maximal kompatiblen Zustandsmengen {S-1, S-2} und {T-1}. Die minimale Überdeckung besteht aus genau diesen beiden Mengen, die von hier an S und T genannt werden. In Abbildung 4.4.4 ist das resultierende Oberflächenmodell dargestellt.

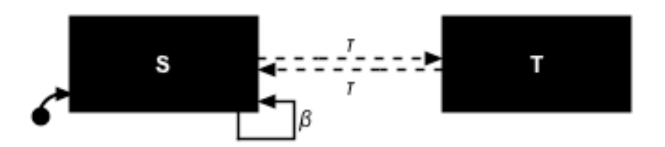


Abbildung 4.4.4: Das vom Reduktionsalgorithmus berechnete Oberflächenmodell des Systems aus Abbildung 4.4.1

Ein Blick auf das unterliegende System zeigt nun, dass vom Startzustand S-1 keine ausgehende Transition nach T-1 führt. Aber der zu S-1 gehörende Zustand des Interfaces ist S, in dem laut Oberflächenmodell angeblich ein automatisches Ereignis τ einen Wechsel nach T auslösen kann. Das heißt, S verkörpert einen *augmenting state,* dem Benutzer wird fälschlicherweise mitgeteilt, dass τ auftreten kann. Solches Verhalten widerspricht direkt den von Heymann und Degani entwickelten Korrektheitskriterien, das Oberflächenmodell aus Abbildung 4.4.4 ist inkorrekt. Die Annahme muss also falsch gewesen sein, woraus Satz 4.4.1 folgt. ■

Das Problem lässt sich beheben, indem eine zusätzliche Bedingung für die Einteilung in Spezifikationsklassen gefordert wird: Benutzergesteuerte Transitionen dürfen niemals zwischen zwei verschiedenen Zuständen derselben Spezifikationsklasse bestehen. Formal ausgedrückt:

Definition 4.4.1: Erweiterung der Definition von Spezifikationsklassen

Sei M die Menge der Zustände, U die Menge der benutzergesteuerten Transitionen und C die Menge der Spezifikationsklassen eines Systems. Zusätzlich zu Definition 3.2.1 gelte: $\forall S_1,S_2 \in M: (((S_1,S_2) \in U \wedge S_1 \neq S_2) \Rightarrow \neg \exists C_1 \in C: (S_1 \in C_1 \wedge S_2 \in C_1))$

Diese Forderung erscheint auf Anhieb sinnvoll, da jede Aktion des Benutzers eine sichtbare Wirkung haben sollte, wie schon in Kapitel 2 herausgearbeitet wurde. Doch zweifelsohne stellt sie eine Einschränkung des Verfahrens dar. Nichtsdestotrotz verhindert sie die Möglichkeit, dass *augmenting states* auftreten, wie im folgenden Teilkapitel gezeigt werden wird.
Unabhängig davon folgt aus dieser Bedingung noch nicht, dass der Reduktionsalgorithmus alle drei Korrektheitskriterien einhält, wie der Beweis des folgenden Satzes zeigt.

Satz 4.4.2

Der Reduktionsalgorithmus gewährleistet nicht die Vermeidung von restricting states.

Beweis von Satz 4.4.2 über Widerspruch: Angenommen, der Algorithmus vermeidet *restricting states.* Dann muss er deren Existenz in allen berechneten Oberflächenmodellen generell ausschließen. Es werde nun das Maschinenmodell in Abbildung 4.4.5 betrachtet. Dieses Modell entspricht bis auf eine Ausnahme dem zweiten Beispiel aus HEYMANN & DEGANI 2006: Während dort eine

ausgehende benutzergesteuerte Transition *ud* aus dem Zustand 42 in den Zustand 32 führt (VGL. HEYMANN & DEGANI 2006, S. 21), ist der Folgezustand hier 51.

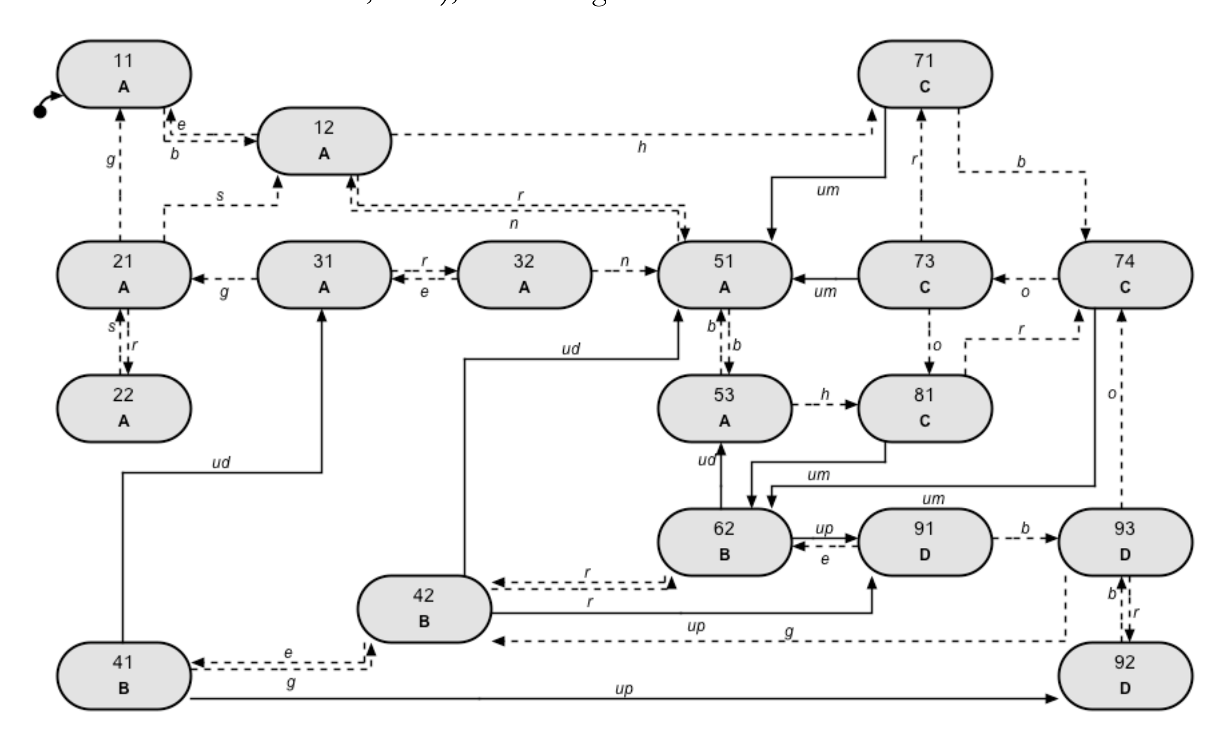


Abbildung 4.4.5: Maschinenmodell eines abstrakten Systems inklusive Spezifikationsklassen

Die Anwendung des Reduktionsalgorithmus sei hier nicht genauer beschrieben. Sie verläuft bis auf die Erzeugung der Transitionen des Oberflächenmodells annähernd genauso, wie sie in HEYMANN & DEGANI 2006 beschrieben wird, liefert insbesondere die gleiche Überdeckung durch maximal kompatible Zustandsmengen. Das resultierende Oberflächenmodell ist in Abbildung 4.4.6 dargestellt.

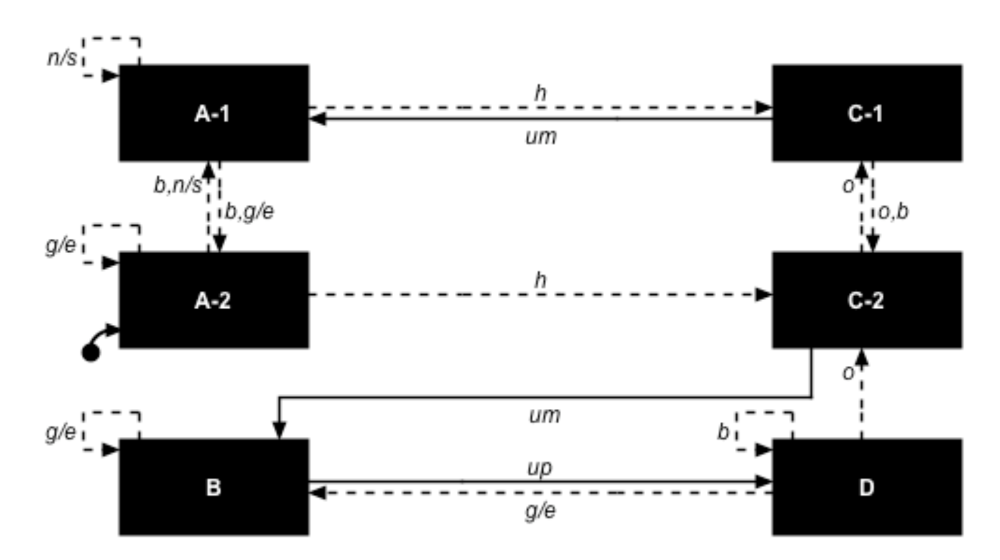


Abbildung 4.4.6: Vom Reduktionsalgorithmus berechnetes Oberflächenmodell des abstrakten Systems aus Abbildung 4.4.5

Es zeigt sich, dass das benutzergesteuerte Ereignis *ud*, welches aus allen internen Zuständen der Spezifikationsklasse **B** einen Wechsel in die Klasse **A** bewirkt, im Oberflächenmodell durch keine Transition repräsentiert wird. Das liegt daran, dass weder A-1 noch A-2 *alle* möglichen Zustände

beinhaltet, die sich mittels *ud* aus einem der Zustände aus **B** erreichen lassen. {31, 51, 53} ist also weder Teilmenge von {12, 21, 22, 31, 32, 51} noch von {11, 21, 22, 31, 32, 53}. Daraus ergibt sich, dass dem Benutzer verschwiegen wird, dass sich sowohl A-1 als auch A-2 durch Auslösen von *ud* in B erreichen lassen. B konstituiert einen *restricting state* und folglich ist auch das Oberflächenmodell aus Abbildung 4.4.6 inkorrekt. Diese Feststellung steht im Widerspruch zur Annahme, woraus Satz 4.4.2 folgt. ■

Bevor die Behebung des soeben gezeigten Fehlers vorgestellt wird, seien noch einmal die in Definition 2.2.1 aufgeführten Korrektheitskriterien untersucht; sie gewährleisten, dass der Benutzer immer ausreichend über den aktuellen Zustand bescheid weiß, dass er das Resultat einer etwaigen Aktion vorausahnen kann und dass ihm keine falschen Versprechungen über mögliche Zustandswechsel gemacht werden. Wie jedoch schon in Kapitel 2.2 angedeutet wurde, ist der Fall nicht abgedeckt, in dem ein System einen Zustandswechsel der Oberfläche hervorruft, den der Benutzer nicht abzusehen vermag. Das folgende Beispiel demonstriert, dass solch eine Situation auftreten kann. In Abbildung 4.4.7 ist ein abstraktes Maschinenmodell samt der Einteilung in Spezifikationsklassen dargestellt, bei dem das Augenmerk auf den Zuständen der Spezifikationsklasse **S** liegen wird.

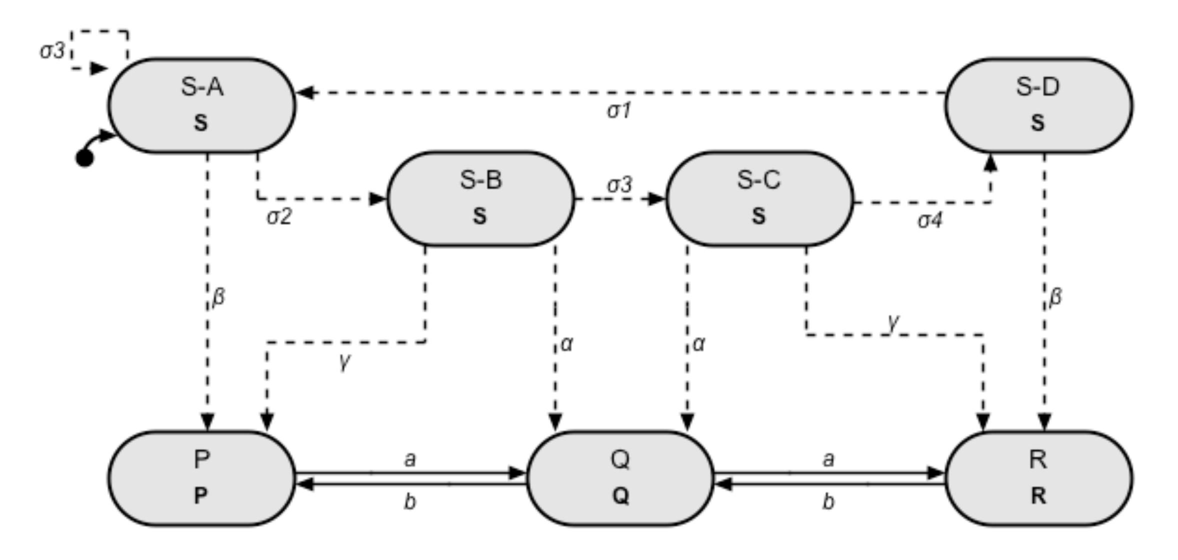


Abbildung 4.4.7: Abstraktes Maschinenmodell eines Systems inklusive Spezifikationsklassen

Auf die Beschreibung des Vorgehens des Reduktionsalgorithmus sei auch hier verzichtet, er liefert das Oberflächenmodell aus Abbildung 4.4.8 auf der folgenden Seite. S-A und S-B wurden zu S-1 zusammengefasst, S-C und S-D zu S-2. Unter anderem wurde außerdem das automatische Ereignis *σ3* internalisiert und zwar deshalb, weil es wie eben keinen Zustand im Oberflächenmodell gibt, der die möglichen Folgezustände S-A und S-C eines Auftretens von *σ3* repräsentiert. Dadurch zeigt das Oberflächenmodell und somit das resultierende User Interface nicht, dass ein Wechsel von S-1 nach S-2 möglich ist. Schlimmer noch: Der Zustand S-2 ist aus Benutzersicht scheinbar unerreichbar.

Diese Gegebenheit widerspricht zwar nicht den Korrektheitskriterien, S-1 verkörpert also keinen *error state, restricting state* oder *augmenting state*. Sie vermindert auch nur bedingt die Bedienbarkeit, da der Benutzer einen Wechsel nach S-2 insofern wahrnimmt, als ihm der gültige Zustand jederzeit

angezeigt wird. Doch sie ist trotzdem alles andere als befriedigend, denn anderenfalls könnte jede andere automatische Transition ebenso gut verschwiegen werden.

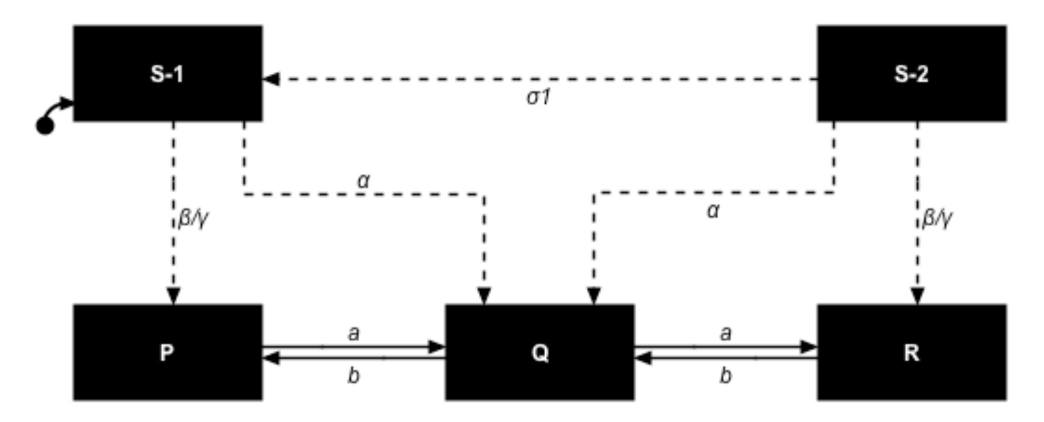


Abbildung 4.4.8: Das vom Algorithmus berechnete Oberflächenmodell des Systems aus Abbildung 4.4.7

Es ist anzumerken, dass aus Gründen der Übersichtlichkeit hier ein Beispiel mit $\sigma 3$ als *self-loop* gewählt wurde. Das Problem kann jedoch auch bei Ereignissen, die ausschließlich Wechsel zwischen Zuständen auslösen, auftreten. Es bedarf also einer zusätzlichen Forderung für das zu generierende Modell, die absichert, dass eine Transition wie die von $\sigma 3$ nicht internalisiert wird. Daher wird nun ein weiteres Korrektheitskriterium eingeführt, das sich der Vermeidung von Zuständen widmet, die mögliche interne Ereignisse verheimlichen, welche einen Wechsel des Zustands des Interfaces nach sich ziehen:

Definition 4.4.1: Erweiterte Korrektheit von User Interfaces

Ein User Interface inklusive aller zugehörigen Hilfsmaterialien ist **erweitert korrekt**, wenn es Definition 2.2.1 genügt und kein wie folgt definierter Zustand existiert:

- **concealing state:** Das System kann einen Zustandswechsel des User Interfaces auslösen, der vom User Interface und den Hilfsmaterialien verschwiegen wird.

Satz 4.4.3:

Der Reduktionsalgorithmus gewährleistet nicht die Vermeidung von concealing states.

Beweis von Satz 4.4.3 über Widerspruch: Angenommen, der Algorithmus vermeidet *concealing states.* Dann hätte er im gerade angeführten Beispiel $\sigma 3$ nicht internalisieren dürfen. Die Annahme muss also falsch sein, was den Satz beweist. ■

Wie kann nun der Reduktionsalgorithmus der Vermeidung von *restricting states* und *concealing states* gerecht werden? Offensichtlich werden Ereignisse fälschlicherweise ausgeblendet, wenn kein *richtiger* Folgezustand ausgemacht wird. Beim ersten Schritt des Festsetzens der Transitionen werden Ereignisse wie eben ud und $\sigma 3$ also gar nicht erst erzeugt. Ins Allgemeine übertragen wird ein ausgehendes Ereignis β von kompatiblen Systemzuständen $S_1, ..., S_k$ bei dem sie repräsentierenden Zustand S_{user} genau dann vernachlässigt, wenn kein Zustand T_{user} des Oberflächenmodells existiert, welcher alle aus β resultierenden Folgezustände $T_1, ..., T_m$ enthält.

Es ist klar, dass eine Transition zwischen S_{user} und allen wie T_{user} beschaffenen Zuständen sinnvoll ist, sofern solche Zustände Teil der minimalen Überdeckung sind. Ansonsten muss aber Abhilfe geleistet werden, um dem Verschwinden von β entgegenzuwirken.
Der Lösungsansatz, der hier verfolgt werden soll und per Option im Software-Tool aktivierbar ist, besteht darin, im Falle des Fehlens eines Zustands T_{user} mit β gelabelte Transitionen von S_{user} aus zu allen Zuständen zu erstellen, die mindestens einen Zustand $T_i \in \{T_1, ..., T_m\}$ beinhalten. Abbildung 4.4.9 verdeutlicht, was für das System aus Abbildung 4.4.7 beim Erzeugen der Transitionen mit $\sigma 3$ passiert.

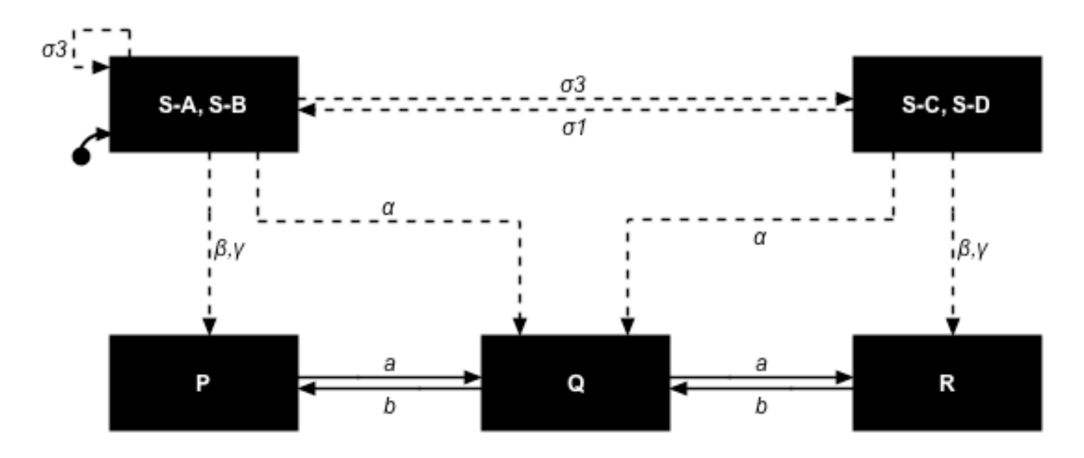


Abbildung 4.4.9: Reduziertes Maschinenmodell des Systems aus Abbildung 4.4.7 nach Erstellung aller Transitionen

Durch diese Maßnahme entsteht selbstverständlich Nicht-Determinismus, der im nächsten Schritt eigentlich gleich wieder beseitigt werden würde. Da dies dem Ziel, mögliche Wechsel von Zuständen sichtbar zu machen, gerade entgegensteht, wird ein anderer Weg gewählt: Jede vorerst mit β bezeichnete Transition bekommt einen eindeutigen Namen, die zugehörigen Ereignisse werden also maskiert. Innerhalb der implementierten Software findet etwa eine Umbenennung in β, β', β'' usw. statt. Dem Benutzer wird also vermittelt, dass es sich um verschiedene Ereignisse handelt. Abbildung 4.4.10 zeigt das resultierende Oberflächenmodell des abstrakten Systems aus Abbildung 4.4.7, in dem der Name eines der Ereignisse $\sigma 3$ durch $\sigma 3'$ ersetzt wurde.

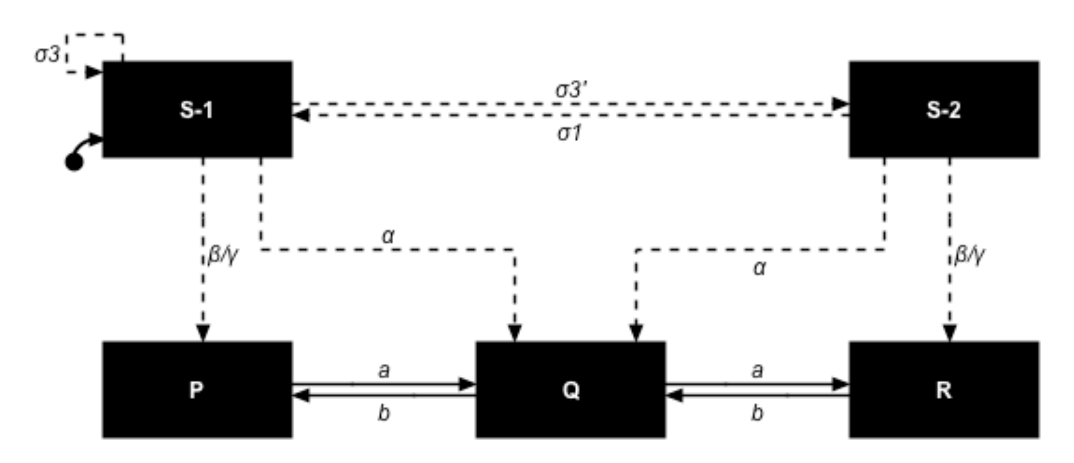


Abbildung 4.4.10: Korrektes und minimales Oberflächenmodell des Systems aus Abbildung 4.4.7

Im folgenden Teilkapitel wird gezeigt werden, dass auf diese Weise das Auftreten von *concealing states* ausgeschlossen wird. Vorher sei aber noch betrachtet, was für Konsequenzen die geänderte Spezifikation des Algorithmus für das Beispiel aus Abbildung 4.4.5 hat, bei dem ein *restricting state* nachgewiesen wurde. Es ergibt sich jetzt folgendes Oberflächenmodell (Abbildung 4.4.11):

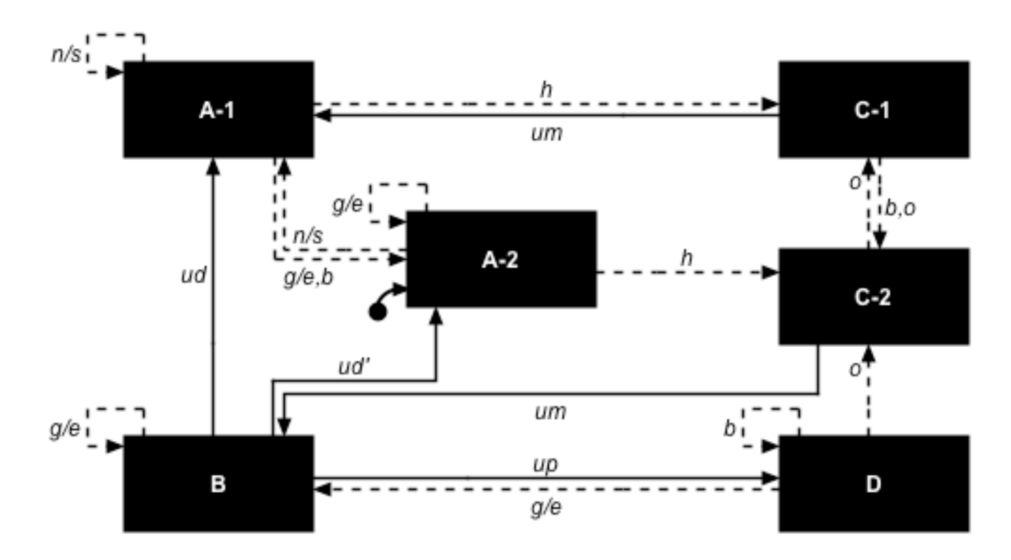


Abbildung 4.4.11: Korrektes und minimales Oberflächenmodell des Systems aus Abbildung 4.4.5

Das benutzergesteuerte Ereignis *ud* wurde also einmal in *ud'* umbenannt. Während das Ändern eines automatischen Ereignisses keinen bedeutungsvollen Einfluss auf das zu entwickelnde User Interface hat, ist das in diesem Fall anders. Denn ein weiteres benutzergesteuertes Ereignis impliziert, dass der Benutzer des Systems eine weitere Aktion tätigen können muss, das heißt, ein zusätzliches Bedienelement erweist sich als obligatorisch. Dieses Resultat ist sicherlich nicht wünschenswert, denn dadurch geht die Minimierung der Anzahl an Zuständen mit der Erhöhung der Anzahl notwendiger Benutzeraktionen einher. Es stellt sich jedoch die Frage, inwiefern es Alternativen für die hier angestellte Korrektur gibt. Diese Frage führt zu dem unterliegenden Problem, das Situationen hervorruft, in denen *concealing states* und *error states* auftreten:
Wenn der einzige Weg, im Schritt der Determinierung der Transitionen solche Zustände zu vermeiden, ist, neue Transitionen zu generieren, dann liegt der Schluss nahe, dass die von einem solchen Zustand repräsentierten Systemzustände *im Grunde* nicht kompatibel sind bzw. dass die Definition von Kompatibilität nicht genügend Bedingungen enthält. Es müssten demnach Einschränkungen getroffen werden, die verhindern, dass resultierende Transitionen des Oberflächenmodells zu so nicht existenten Zuständen führen. Ein solches Konzept hätte aber die Überarbeitung der Berechnung maximal kompatibler Zustandsmengen zur Folge, was den Umfang dieser Arbeit übersteigt.
Abgesehen davon ist der oben angeführte Lösungsansatz keineswegs falsch, im Gegenteil: Die von dem geänderten Reduktionsalgorithmus berechneten Oberflächenmodelle genügen allen geforderten Korrektheitskriterien. Diese Behauptung soll gleich bewiesen werden, ebenso wie die Minimalität der sich ergebenden Modelle. Zudem macht die Einführung der umbenannten Transitionen das besprochene Problem im Oberflächenmodell erkennbar und bildet damit eine sinnvolle Basis für das User Interface Design.

4.5 KORREKTHEIT DES REDUKTIONSALGORITHMUS

In diesem Teilkapitel geht es darum zu zeigen, dass der geänderte Reduktionsalgorithmus korrekt ist, also stets ein korrektes und minimales Oberflächenmodell zu einem gegebenen Modell der Maschine und der formalisierten Aufgabenspezifikation berechnet.

Satz 4.5.1:

> **Der geänderte Reduktionsalgorithmus gewährleistet unter der erweiterten Definition von Spezifikationsklassen (Definition 4.4.1) die Einhaltung aller vier geforderten Korrektheitskriterien.**

Beweis von Satz 4.4.2: Es ist zu beweisen, dass für jedes vom Reduktionsalgorithmus berechnete Oberflächenmodell die vier folgenden Aussagen gelten: (1) Das Oberflächenmodell enthält keinen *error state,* (2) es enthält keinen *restricting state*, (3) es enthält keinen *augmenting state* und (4) es enthält keinen *concealing state.*

Beweis von (1): Allgemein ist klar, dass nur kompatible Zustände des Maschinenmodells im Oberflächenmodell durch den gleichen Zustand repräsentiert werden können und dass Zustände verschiedener Spezifikationsklassen niemals kompatibel sind. Daher reicht es zu zeigen, dass von zwei kompatiblen Zuständen aus keine Eingabesequenz in Zustände führt, die nicht derselben Spezifikationsklasse angehören. Dies wird mittels Induktion über die Länge n aller möglichen Ereignissequenzen bewiesen. Sei $n = 1$: Der Algorithmus deklariert Zustände als inkompatibel, wenn ein gemeinsames Ereignis in Zustände verschiedener Spezifikationsklassen führt. Schalten einer einzigen Transition kann in kompatiblen Zuständen daher nicht in einem *error state* münden. Es folgt die Induktionsvoraussetzung: Für ein beliebiges, aber festes n ist die Existenz von *error states* ausgeschlossen. Seien also weiter S und T zwei kompatible Zustände des Maschinenmodells und $\beta_1...\beta_{n+1}$ eine von S und T aus definierte Ereignissequenz der Länge $n+1$. Zu zeigen ist, dass $\beta_1...\beta_{n+1}$ niemals in Zuständen verschiedener Spezifikationsklassen resultiert: Die Sequenz $\beta_1...\beta_n$ schalte in ein Folgezustandspaar (S', T'). Nach Induktionsvoraussetzung ist der S' und T' repräsentierende Zustand des Oberflächenmodells kein *error state*. Zu dem Zeitpunkt, an dem der Reduktionsalgorithmus die Ereignisfolge $\beta_1...\beta_n$ durch Ersetzen der Zustandspaare in der *merger table* simuliert hat, wurde dies für S' und T' zumindest auch für β_{n+1} bereits getan. Da nun S und T kompatibel sind, müssen auch S' und T' vom Algorithmus als kompatibel berechnet worden sein. Das heißt insbesondere, dass β_{n+1} von S' und T' aus nicht in Zustände verschiedener Spezifikationsklassen führt. Es folgt, dass für keine definierte Ereignissequenz der Länge $n+1$ ein *error state* erreicht wird.

Beweis von (2) und (4): Wie gerade gezeigt, weisen die generierten Oberflächenmodelle keine *error states* auf, der aktuelle Zustand des unterliegenden Systems ist dem Benutzer also jederzeit bekannt. Darüber hinaus werden weder benutzergesteuerte noch automatische Transitionen vom geänderten Reduktionsalgorithmus internalisiert, sofern diese einen Zustandswechsel bedingen und keine nicht-deterministische Situation hervorrufen. Das Entfernen einer *self-loop*-Transition kann offensichtlich zu keinem *concealing state* führen, benutzergesteuerte *self-loops* werden ohnehin nicht entfernt. Sei also β eine nicht-deterministische ausgehende Transition eines Zustands S_{user} des Oberflächenmodells, deren entsprechenden Transitionen im Maschinenmodell zu Zuständen

$S_1, ..., S_k$ führe. β wird vom geänderten Reduktionsalgorithmus nur dann internalisiert, wenn es eine andere ebenfalls mit β gelabelte, ausgehende Transition von S_{user} gibt, die zu einem Zustand des Oberflächenmodells führt, der mindestens auch die Zustände $S_1, ..., S_k$ repräsentiert. Daher ist die Existenz von *restricting states* und *concealing states* ausgeschlossen.

Beweis von (3): Die in Definition 4.4.1 eingeführte Bedingung impliziert, dass nur *automatische* Ereignisse $\tau_1, ..., \tau_m$ Wechsel zwischen kompatiblen Zuständen $S_1, ..., S_k$ einer Spezifikationsklasse C_1 auslösen können. Nach Definition der Kompatibilität kann kein Ereignis $\tau_j \in \{\tau_1, ..., \tau_m\}$ von einem der Zustände $S_i \in \{S_1, ..., S_k\}$ in eine andere Spezifikationsklasse C_2 führen. Daher könnten alle Ereignisse τ_j internalisiert werden. Es ändert sich somit aus Sicht des Benutzers nichts, während das System in einem der Zustände S_i ist. Daraus folgt, dass der Benutzer nicht unterscheiden muss, in welchem S_i ein internes Ereignis $\beta \notin \{\tau_1, ..., \tau_m\}$ auftritt, das einen Wechsel in eine andere Spezifikationsklasse auslöst. Außerdem gilt nach Definition, dass in allen S_i die gleichen benutzergesteuerten Ereignisse ausgelöst werden können. Sei nun S_{user} der Zustand des Oberflächenmodells, welcher die Zustände $S_1, ..., S_k$ repräsentiert. Da belanglos ist, in welchem S_i ein wie oben beschriebenes automatisches Ereignis β auslösbar ist, und alle S_i die gleichen benutzergesteuerten Ereignisse ermöglichen, kann S_{user} niemals einen *augmenting state* konstituieren, denn nach Algorithmus werden im Oberflächenmodell automatische Ereignisse, die nicht mindestens in einem S_i aktiv sind, ebenso wenig aufgeführt wie benutzergesteuerte Ereignisse, die nicht in allen Zuständen $S_1, ..., S_k$ auslösbar sind.

Aus den Beweisen von (1), (2), (3) und (4) folgt unmittelbar die Korrektheit von Satz 4.5.1. ■

Satz 4.5.2

Die Anzahl an Zuständen und Vorkommen von Ereignissen des vom geänderten Reduktionsalgorithmus berechneten Oberflächenmodells ist minimal hinsichtlich der Erfüllung der geforderten Korrektheitskriterien.

Beweis von Satz 4.4.3: Zu beweisen sind die folgenden Aussagen: (1) Die Anzahl an Zuständen ist minimal und (2) die Anzahl an Vorkommen von Ereignissen ist minimal. Bei der zweiten Aussage wird auf die Beachtung aller *self-loop*-Transitionen verzichtet, da es vom konkreten Fall abhängig ist, ob sich deren Abbildung im Oberflächenmodell als notwendig erweist. Aus demselben Grund wird nicht dazwischen unterschieden, ob eine Transition mit einem oder mehreren Ereignissen beschriftet ist. Daher reicht es zu zeigen, dass die Anzahl aller zwischen Zuständen bestehenden Transitionen minimal ist.

Beweis von (1) über Widerspruch: Sei k die Anzahl der Zustände des Oberflächenmodells, die der Reduktionsalgorithmus berechnet hat. Angenommen, es gibt ein Oberflächenmodell mit höchstens k-1 Zuständen, das korrekt ist. Dann gilt zumindest eine der folgenden Aussagen: (a) Die vom Algorithmus ermittelte Überdeckung der maximal kompatiblen Zustandsmengen ist nicht minimal. (b) Mindestens zwei Zustände des Maschinenmodells hätten entgegen der Berechnung des Algorithmus zusammengefasst werden können. Der Algorithmus vergleicht alle möglichen Überdeckungen, also kann Aussage (a) nicht wahr sein. Wenn die Annahme stimmt, hat der Algorithmus somit fälschlicherweise ausgeschlossen, dass zwei Zustände S_1 und S_2 nicht unterschieden werden müssen. Da aus den Korrektheitskriterien unmittelbar zu entnehmen ist, dass kompatible Zustände der gleichen Spezifikationsklasse angehören und die gleichen Aktionen

des Benutzers erlauben müssen, kann dies nur im Resolutionsprozess passiert sein. Dort werden S_1 und S_2 aber nur dann als inkompatibel markiert, wenn eine von ihnen aus definierte Ereignissequenz in inkompatiblen Zuständen resultiert. In diesem Fall müssen zur Vermeidung von *error states* jedoch auch S_1 und S_2 inkompatibel sein. Es folgt, dass S_1 und S_2 nicht in einen Zustand abstrahiert werden dürfen. Auch Aussage (b) ist also falsch, was im Widerspruch zur Annahme steht.

Beweis von (2) über Widerspruch: Angenommen, die Anzahl zwischen Zuständen bestehender Transitionen ist nicht minimal. Dann muss es eine mit $\beta_1, ..., \beta_k$ gelabelte Transition geben, die überflüssig für die Erfüllung der Korrektheitskriterien ist. Diese Transition führe von einem Zustand S des Oberflächenmodells zu einem Zustand T. Die Transition ist entweder (a) benutzergesteuert oder (b) automatisch. Wird die Transition im Fall (a) entfernt, so hat der Benutzer keine Information darüber, dass das Auslösen eines der Ereignisse $\beta_i \in \{\beta_1, ..., \beta_k\}$ in S einen Wechsel nach T auslösen kann. S stellt damit einen *restricting state* dar. Wird die Transition im Fall (b) entfernt, so hat der Benutzer entsprechend keine Information darüber, dass ein internes Ereignis β_i in S einen Zustandswechsel nach T bewirken kann. S verkörpert somit einen *concealing state*. Die Annahme muss also falsch gewesen sein, was zu zeigen war.

Aus den Beweisen von (1) und (2) folgt unmittelbar die Korrektheit von Satz 4.5.2. ■

Satz 4.5.3:

Der geänderte Reduktionsalgorithmus ist unter der erweiterten Definition von Spezifikationsklassen (Definition 4.4.1) erweitert korrekt (Definition 4.4.2).

Beweis von Satz 4.5.3: Die Korrektheit des Reduktionsalgorithmus folgt direkt aus den Beweisen von Satz 4.5.1 und Satz 4.5.2. ■

Der Korrektheitsbeweis zeigt, dass das in den vorigen Teilkapiteln beschriebene Verfahren eine Möglichkeit liefert, das Design eines Interfaces formal durchzuführen. Ebenso rechtfertigt er die Bezeichnung der in Kapitel 4.3 entwickelten Oberflächenmodelle als „korrekt und minimal". An dieser Stelle angekommen stellt sich letzten Endes die Frage, inwiefern der Algorithmus auf realistische Systeme anwendbar ist. Abschließend beschäftigt sich Kapitel 4.6 deshalb mit den Grenzen des Verfahrens, wobei insbesondere die *worst-case*-Laufzeit des Algorithmus eine Rolle spielen wird.

4.6 GRENZEN DER ANWENDBARKEIT DES VERFAHRENS

Eine erste Einschränkung, die im Rahmen dieser Arbeit herausgearbeitet wurde, ist bereits durch die Eingabe des Algorithmus gegeben. Einerseits wird gefordert, dass das zu reduzierende System in Form eines Modells vorliegen muss, welches eine Spezifikation von Zuständen und diskreten Ereignissen voraussetzt. Andererseits – und dies wiegt im Zweifelsfall noch schwerer – wird eine Formalisierung der Aufgabenspezifikation in einer sehr konkreten Form benötigt; die Einteilung in Spezifikationsklassen impliziert, dass die Entwickler grundsätzlich entscheiden können müssen, welche internen Zustände ein und denselben Modus repräsentieren.

Dennoch ist ein nicht unwesentlicher Fortschritt ersichtlich: Bezogen auf die anzuzeigende Information stellt die Partition der Zustände den letzten Eingriff des Menschen während des Prozesses des Designs eines Interfaces dar, was nach dem eben angestellten Beweis der Korrektheit als offensichtlicher Vorteil zu deklarieren ist. Außerdem wird die vorbereitende Modellierung von Abläufen eines Systems heutzutage mehr und mehr zum Standard. Dass diese genau in der hier vorgestellten Weise durchgeführt wird, ist hinzukommend nicht verpflichtend: „As such, the approach, methodology, and algorithm proposed here can be extended to other discrete event formalisms such as Petri-nets and Statecharts, as well as to hybrid systems models that have both continuous and discrete behaviors“ (HEYMANN & DEGANI 2006, S. 25).

Unabhängig von den in Kapitel 4.4 besprochenen und nicht völlig behobenen Schwachstellen im Verfahren, ist ein Hauptproblem des Reduktionsalgorithmus seine Laufzeit im schlechtesten Fall. Angenommen etwa, ein Maschinenmodell beinhalte n Zustände, welche alle einer einzigen Spezifikationsklasse angehören und dieselben Benutzeraktionen ermöglichen (über die Frage des Sinns eines solchen Systems sei hier hinweggesehen). Dann sind alle Zustände in jedem Fall paarweise miteinander kompatibel. Folgerichtig würde der Reduktionsalgorithmus eine maximal kompatible Zustandsmenge der Größe n berechnen. Die Methoden zum Finden kompatibler Zustandsmengen haben zwar für sich nur polynomielle Laufzeit sehr niedrigen Grades (ebenso übrigens die für das Festsetzen der Transitionen). Da der Algorithmus für das beschriebene Modell aber in jedem rekursiven Schritt jeweils alle Teilmengen einer bestimmten Größe $m < n$ miteinander vergleichen muss, folgt exponentielle Laufzeit. Es ist klar, dass sich ähnliche Fälle auch in realistischeren Modellen ergeben können. Auch die Suche nach einer minimalen Überdeckung bedeutet wie gesagt für jede Spezifikationsklasse das Testen jeder möglichen Kombination an maximal kompatiblen Zustandsmengen, die aus der Klasse hervorgehen. Zweifelsohne können also die notwendigen Berechnungen für sehr große Systeme mit tausenden an Zuständen untragbar werden (VGL. AUCH HEYMANN & DEGANI 2006, S. 25).
Wie dem auch sei, es kann behauptet werden, dass die Laufzeit im Allgemeinen trotzdem *theoretisch* nicht schlecht ist, was auf den ersten Blick überraschend klingen mag, sich aber begründen lässt: Die *worst-case*-Laufzeit ist nämlich nicht etwa durch die Anzahl der Zustände eines Systems nach oben abgegrenzt oder schlimmer noch, durch die Anzahl der Transitionen (es ist leicht zu überlegen, dass ein normales System viel mehr Transitionen als Zustände hat). Sondern die obere Schranke definiert sich durch die Größe der größten Spezifikationsklasse. Es lassen sich generell kaum genaue Annahmen über die Kardinalität der größten Klasse im Vergleich zu der der Zustandsmenge machen; sicherlich kann jedoch ausgesagt werden, dass sie bei komplexeren Systemen um ein Vielfaches geringer ist.
Darüber hinaus haben die Beispiele verdeutlicht, dass in der Regel die Kompatibilität vieler Zustandspaare schnell ausgeschlossen werden kann. Im Gegensatz zu der intuitiven Ableitung einer Benutzungsschnittstelle ermöglicht das Verfahren von Heymann und Degani somit, einen Großteil des Betrachtungsgegenstands bereits zu einem frühen Zeitpunkt vernachlässigen zu können und erlaubt es dadurch, auch für viele realistische Systeme beweisbar korrekte Interfaces zu generieren; „the reduction algorithm [...] has been successfully applied to machine models with more than 500 internal states, It may be possible, by improving the efficiency of our algorithm, to reduce even larger machines“ (HEYMANN & DEGANI 2006, S. 25).

Ein letzter Punkt, der hier angesprochen werden soll, ist die Frage nach der Angemessenheit des vom Reduktionsalgorithmus generierten Oberflächenmodells. Da ein allgemeines und im Grunde rein mathematisches Verfahren angewendet wird, besteht nicht zwangsläufig ein logischer Zusammenhang zwischen zusammengefassten Zuständen und deren Bedeutung innerhalb des Systems. „The proposed reduction procedure generates interfaces that are not necessarily intuitive or easily correlated with the underlying system“ (HEYMANN & DEGANI 2002 I, S. 30). Vor allem entspricht es etwa nicht der Vorstellung, die ein Benutzer üblicherweise von einem technischen Gerät oder Programm hat, dass ein interner Zustand durch mehr als einen Zustand des Interfaces repräsentiert werden kann. Diese Tatsache zeigt den essentiellen Unterschied zwischen dem menschlichen Abstraktionsprozess und dem hier vorgestellten Verfahren. Doch die steigende Komplexität aktueller und zukünftiger Systeme und die gleichzeitig steigenden Sicherheitsansprüche an selbige zeigen, dass ein Wechsel zu einem automatisierten Designprozess früher oder später kaum vermeidbar sein wird (VGL. HEYMANN & DEGANI 2002 I, S. 30).

Mit dieser Konklusion soll die Beschäftigung mit der Generierung korrekter und minimaler Oberflächenmodelle zum Abschluss kommen. Es wurde herausgestellt, dass das Verfahren von Heymann und Degani inklusive der genannten Verbesserungen einen innovativen und nützlichen Lösungsweg für die Erzeugung von Interfaces darstellt, der jedoch Aspekte aufweist, die es noch zu evaluieren gilt. Das Vorgehen des Reduktionsalgorithmus wurde hinreichend analysiert und an Beispielen veranschaulicht.

Wie bereits mehrfach angesprochen, wurde im Rahmen dieser Arbeit ein Software-Tool implementiert, das den Algorithmus umsetzt. Da das Programm außerdem einen graphischen Editor zur Erstellung von Maschinenmodellen beinhaltet, kann das Verfahren damit auch praktisch nachvollzogen werden. Die technische Seite und die Funktionalitäten des Programms sind ebenso Thema des nächsten und inhaltlich letzten Kapitels wie eine Anleitung, die den Einstieg in die Bedienung des Editors anhand eines exemplarischen Ablaufs erleichtern wird.

5 ENTWICKLUNG EINER SOFTWARE ZUR ERZEUGUNG VON OBERFLÄCHENMODELLEN

„If there is an available path that the user can take – no matter how unlikely – there will be someone [...] that will take it!“

(DEGANI 2004, S. 127)

5.1 TECHNISCHER HINTERGRUND – ARCHITEKTUR UND INTERNE ABLÄUFE DER SOFTWARE

In diesem Kapitel wird das im Rahmen dieser Arbeit implementierte Software-Tool vorgestellt. Es wurde vollständig in der Programmiersprache *Java* (Version 1.4.2) entwickelt und ermöglicht daher plattformunabhängige Benutzung. Das Programm findet sich auf der dieser Arbeit beigefügten CD-ROM als ausführbare *JAR*-Datei, außerdem finden sich dort der Quellcode und weitere Materialien. Für eine genaue Beschreibung der Inhalte sei auf den Anhang verwiesen.
Ehe es gleich darum gehen wird, die einzelnen Funktionalitäten des Programms zu erläutern und die Benutzung anschließend anhand eines beispielhaften Wegs zu zeigen, widmet sich dieses Teilkapitel dem technischen Hintergrund, behandelt dabei die wesentlichen Klassen, Strukturen und Abläufe der Software. Von genaueren Darstellungen der einzelnen vorbereitenden Schritte des Softwareentwicklungsprozesses wird hingegen abgesehen.

Sei nun das vereinfachte Klassendiagramm des Programms in Abbildung 5.1.1 auf der folgenden Seite betrachtet. Der Übersichtlichkeit halber wurden alle Attribute und Methoden ausgeblendet. Weiterhin wurden einige Übergangsklassen ausgelassen, welche einzelne Komponenten der Bedienungsoberfläche sowie Dialoge betreffen, ebenso wie mehrere Datenhaltungsklassen, deren Bedeutung für das Verstehen der Strukturen wenig relevant ist.
Das implementierte System hält sich an das Prinzip der *3-Schichten-Architektur*, trennt also die drei Bereiche Datenspeicherung *(entity)*, Benutzerausgabe *(boundary)* und die Ausführung wesentlicher Systemoperationen *(control)* weitestgehend voneinander. Dabei wurden die Schnittstellen zwischen diesen Bereichen minimal gehalten; der Zugriff auf systeminterne Daten findet ausschließlich über die Kontrollklasse **DataControl** statt. Sie fungiert in gewisser Hinsicht als *Server*, bietet also die Verwaltung der Daten als Dienst an, ohne von sich aus mit der jeweiligen Kontrollklasse zu kommunizieren. Bezüglich der Ausgabe kennt keine Kontrollklasse eine andere Übergangsklasse als **MainGUI** – jede Übergangsklasse außer **MainGUI** ließe sich daher austauschen ohne die anderen Schichten zu beeinflussen. Die **MainGUI** liefert das Grundgerüst der Oberfläche selbst und gibt ansonsten den Großteil der Aktualisierungsaufträge an die einzelnen Klassen der Übergangsschicht weiter.

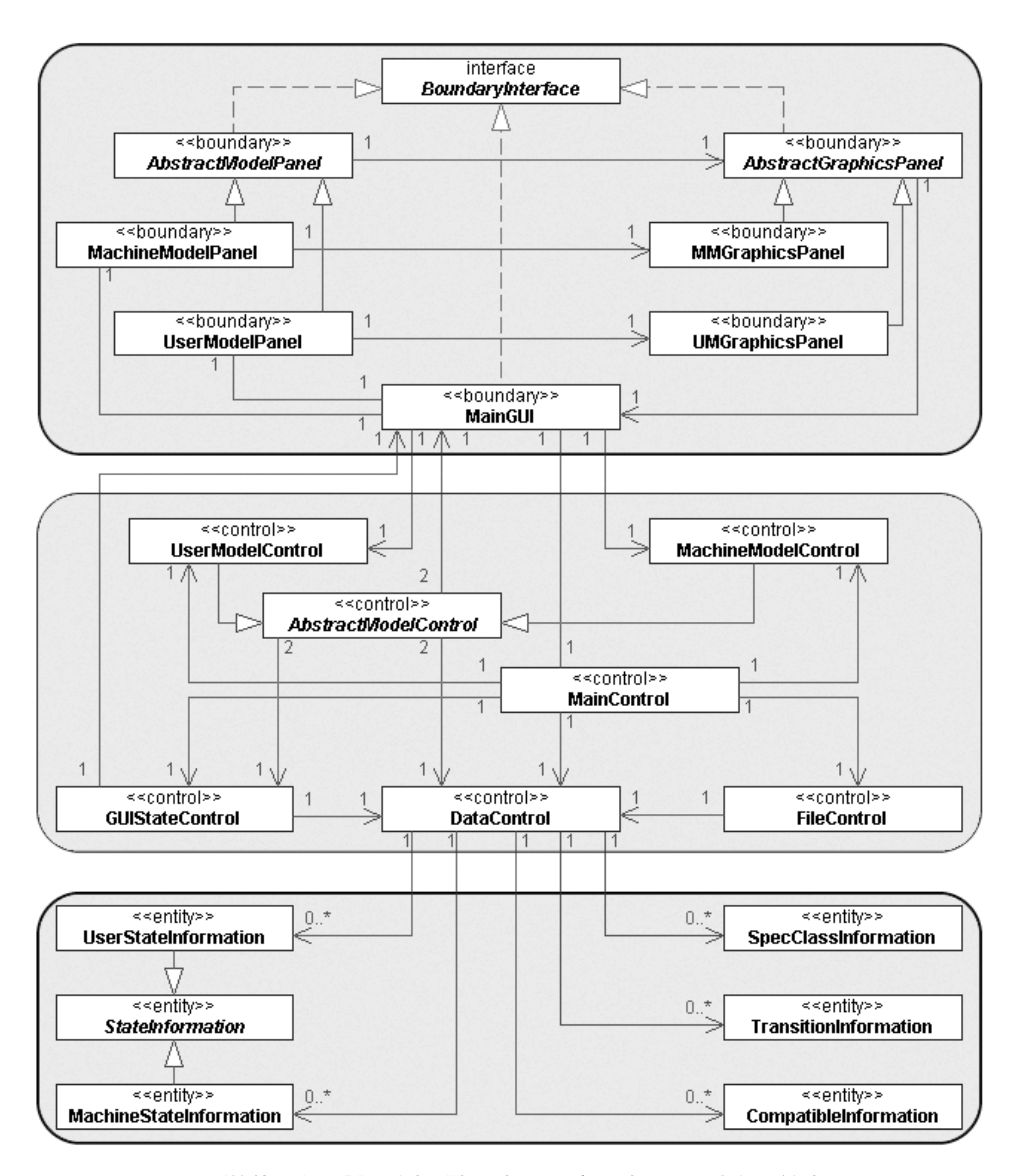


Abbildung 5.1.1: Vereinfachtes Klassendiagramm des implementierten Software-Tools

Beim Start des Programms erzeugt die *main*-Methode in der Klasse **MainControl** die **MainGUI** sowie alle im Klassendiagramm sichtbaren Kontrollklassen. Sie ist außerdem für die nötigen Berechnungen vor dem Lesen oder Schreiben externer Daten verantwortlich, deren Ergebnisse dann an die **FileControl** weitergeleitet werden, von wo aus exklusiv nach außen kommuniziert wird. Diese speichert und lädt Daten über die Modelle in Form so genannter *properties*-Dateien. Die **MainGUI** erstellt bei ihrer Initialisierung die Komponenten der Oberfläche. Neben den hier nicht dargestellten Klassen für Menü, Tool-Bar und weitere Steuerungselemente sind das vor allem das **MachineModelPanel** und das **UserModelPanel**. Diese beiden Klassen ermöglichen den Zugriff des Benutzers auf die Modelle und beinhalten jeweils eine Zeichenfläche

(**MMGraphicsPanel**, **UMGraphicsPanel**) für die Anzeige der Modelle. Alle Panels erben von abstrakten Klassen, die grundlegende Funktionen anbieten. Allgemein implementiert jede Übergangsklasse das Interface **BoundaryInterface**, das Methoden für das Setzen des Zustands und sprachspezifische Ausgabewerte aufweist (vgl. unten).

Zwei zentrale Klassen im System sind die **MachineModelControl** und die **UserModelControl**. Erstere reagiert auf Ereignisse, die durch Benutzereingaben am Maschinenmodell auftreten und ihr von den Übergangsklassen mitgeteilt werden, und berechnet als Folge alle für die Erstellung des Maschinenmodells wichtigen Informationen. Zweitere übernimmt einerseits dieselben Aufgaben für das Oberflächenmodell. Andererseits ist sie insbesondere diejenige, in welcher der Reduktionsalgorithmus ausgeführt wird, in der also das Verfahren von Heymann und Degani umgesetzt ist. Beide Klassen kommunizieren mit der **DataControl** zur Erzeugung, Veränderung, Löschung oder Ermittlung von Daten über Zustände (Kindklassen von **StateInformation**) und Transitionen (**TransitionInformation**) sowie Spezifikationsklassen (**SpecClassInformation**) und kompatiblen Zustandsmengen (**CompatibleInformation**) und beordern danach die **MainGUI**, etwaige Aktualisierungen der Oberfläche vorzunehmen. Sie erben von der abstrakten Oberklasse **AbstractModelControl**, in der unter anderem von beiden benötigte Methoden wie die Berechnung der Punkte einer Transition oder die Selektion und Bewegung von Objekten des Modells enthalten sind.

Die letzte noch zu nennende Kontrollklasse des Programms ist die **GUIStateControl**. Sie ist zuständig für das Aktivieren und Deaktivieren von Bedienelementen der Benutzeroberfläche. Dieser unterliegt nämlich ein Zustandsautomat, mittels dem nach jeder Aktion herausgefunden wird, welche Funktionen für den Benutzer als nächstes sinnvoll sind. Die **GUIStateControl** erfragt für das Aktualisieren des Zustands bestimmte Daten von der **DataControl** und gibt den zu setzenden Zustand an die Übergangsklassen weiter.

Exemplarisch werde hier noch der Ablauf beim Generieren des Oberflächenmodells beschrieben, um einen flüchtigen Einblick in die Funktionsweise des Systems zu geben: Betätigt der Benutzer die zugehörige Schaltfläche, so wird von der **MainGUI** aus die entsprechende Methode in der **UserModelControl** aufgerufen. Die **UserModelControl** erfragt daraufhin Daten bei der **DataControl** und führt den Reduktionsalgorithmus aus. Während der einzelnen Schritte des Algorithmus werden über die **DataControl** mehrere Male neue Daten, wie die der Zustände des Oberflächenmodells (**UserStateInformation**), erstellt. Anschließend wird die **MainGUI** zur Anzeige des Modells in der Benutzeroberfläche beauftragt, deren Zustand zuletzt die **GUIStateControl** anpasst.

Dieses Teilkapitel sollte lediglich einen Überblick über die technische Seite des Software-Tools geben. Ausgewählte Teile der Umsetzung des Reduktionsalgorithmus wurden außerdem schon in Kapitel 4 in Form von Pseudocode dargestellt. Eine fundierte Erörterung der Implementierung ist nicht Inhalt der Arbeit.

Die Funktionalitäten des Programms werden im Folgenden anhand der Bedienungsoberfläche vorgestellt. Dabei werden alle wesentlichen Steuerungsmöglichkeiten und Anzeigen, welche für die Benutzung des Programms wichtig sind, aufgelistet. Ihr konkreter Einsatz wird zumindest partiell hinterher in Kapitel 5.3 besprochen.

5.2 ÜBERBLICK ÜBER DIE FUNKTIONEN DES GRAPHISCHEN EDITORS

Abbildung 5.2.1 zeigt die gesamte Bedienungsoberfläche der erstellten Software unter *Mac OS X*. Das Aussehen der Oberfläche kann je nach Betriebsystem variieren. Bereits zu Beginn sei darauf hingewiesen, dass hier die englische Version beschrieben wird. Das System wird standardmäßig mit der eingestellten Sprache des Betriebssystems gestartet, über den Menüpunkt Language kann aber zu jeder Zeit zwischen Deutsch und Englisch umgeschaltet werden. Andere Sprachen sind nicht eingebunden, ließen sich aber ohne Schwierigkeiten in die Software integrieren.

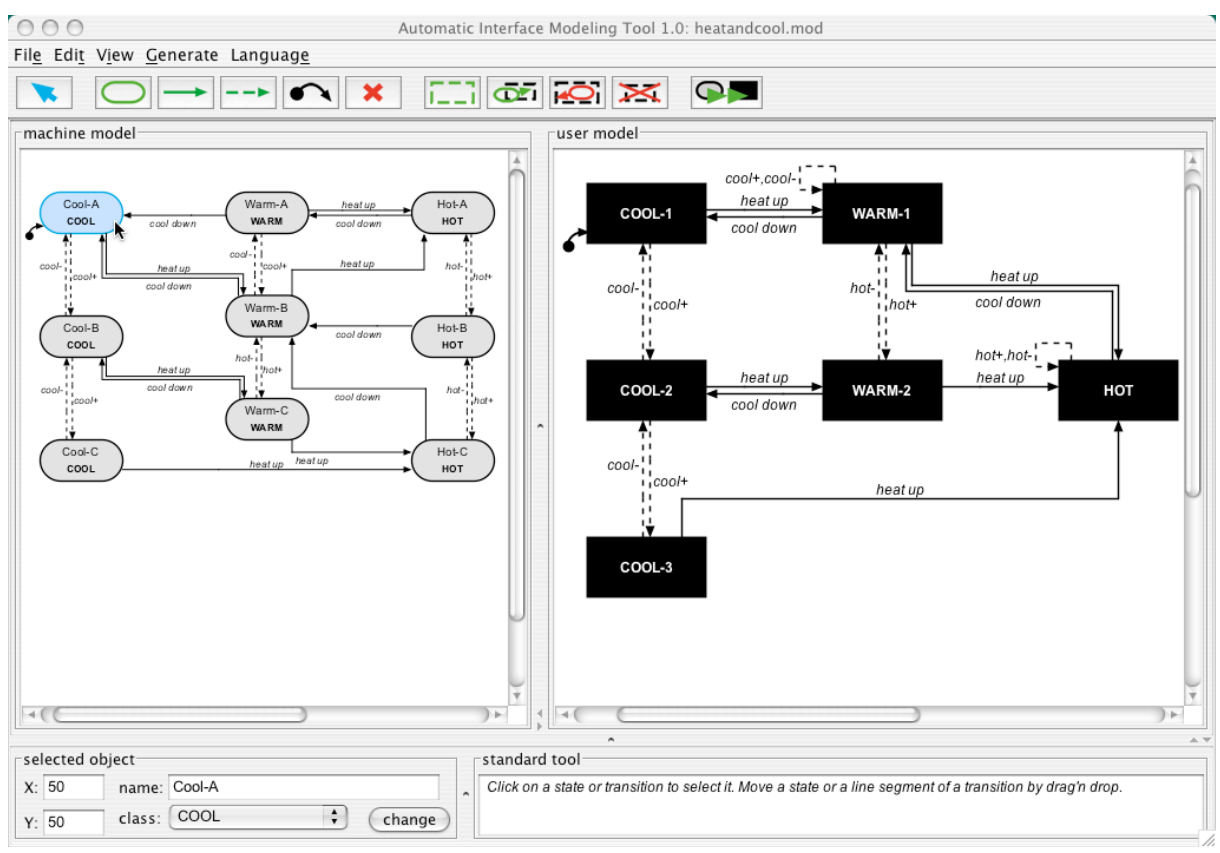


Abbildung 5.2.1: Die gesamte Oberfläche des Software-Tools auf Englisch unter Mac OS X

Die Oberfläche besteht nun grundsätzlich aus sechs Bereichen: Oben finden sich Menü und Tool-Bar des Programms; die darin enthaltenen Funktionalitäten werden noch beschrieben, sie lassen sich alternativ auch über Tastaturkürzel aktivieren. Dabei werden die Kürzel an das verwendete Betriebssystem angepasst, denn während etwa die Standardmaskierung unter *Windows* oder *Linux* mittels der Taste „*[ctrl]*“ geschieht, findet diese bei *Mac OS X* nicht durch „*[ctrl]*“ sondern durch „*[]*“ statt.

In der Mitte gibt es links einen Bereich für das Maschinenmodell *(machine model)* und rechts einen für das Oberflächenmodell *(user model)*. Diese beiden Arbeitsflächen sind in ein *split panel* eingebettet, so dass sich ihre Größe bei Bedarf mittels der Regler zwischen den Modellen anpassen oder eins von beiden vorübergehend völlig ausblenden lässt.

Der untere Teil der Oberfläche beinhaltet auf der linken Seite eine Anzeige für Informationen des selektierten Zustands bzw. der selektierten Transition *(selected object)*, welche auch Eingaben

des Benutzers erlaubt. Aktuell ist in der Abbildung der Zustand Cool-A des Maschinenmodells selektiert (erkennbar an der blauen Einfärbung), dessen Koordinaten in der Fläche sowie Name und Klassenzugehörigkeit in den Textfeldern erscheinen. Rechts von diesem Bereich ist ein Ausgabefeld, das eine Kurzbeschreibung der Funktionsweise des ausgewählten Werkzeugs in sich trägt, hier die des *standard tools* (im Tool-Bar ebenfalls blau eingefärbt), welches dem Selektieren und Bewegen von Objekten dient.

Das Programm ermöglicht das Laden und Speichern von Modellen in einem eigenen Dateiformat **.mod*. Die Daten von Maschinen- und Oberflächenmodell werden dabei – so letzteres vorhanden ist – in der gleichen Datei aufgeführt. Weiter lassen sich die Modelle einzeln als Bild des Typs *PNG* exportieren. Diese und andere Dateifunktionen finden sich unter dem Menüpunkt File.
Für die Bearbeitung des Maschinenmodells stehen zehn Werkzeuge zur Verfügung, die sich sowohl über das Menü (Edit) als auch über den Tool-Bar (alle Schaltflächen außer der ganz rechts) anwählen lassen. Sie reichen von der Erzeugung von Zuständen, benutzergesteuerten und automatischen Transitionen über das Festlegen des Startzustands und das Löschen von Objekten bis hin zur Erstellung von Spezifikationsklassen und dem nachträglichen Anpassen oder Löschen selbiger. Ein Klick auf einen der Buttons wählt das entsprechende Werkzeug aus; unten rechts werden dazu kontextsensitive Informationen angezeigt. Weitere Erläuterungen zur Verwendung dieser Funktionen folgen im nächsten Abschnitt.
Die Darstellung der Modelle kann über den Menüpunkt View individualisiert werden. So lässt sich die Größe der Labels von Zuständen, Transitionen und Klassen genauso wie die der Arbeitsflächen der Modelle über einen Dialog (Abbildung 5.2.2) ändern. Auch kann die Anzeige der Modelle unter View einzeln skaliert werden. In Abbildung 5.2.1 ist das Maschinenmodell beispielsweise verkleinert worden, um es im Gesamten betrachten zu können.

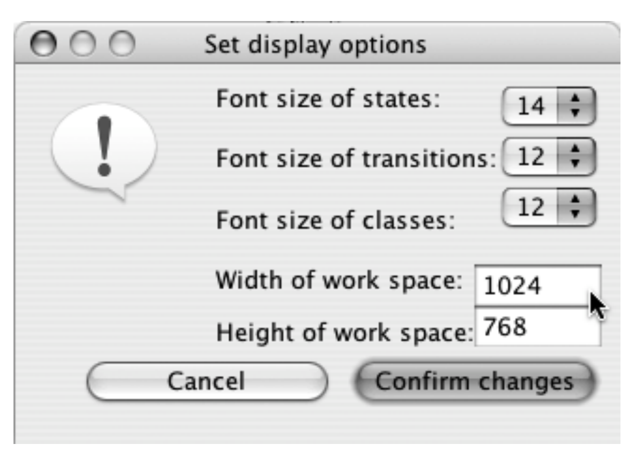


Abbildung 5.2.2: Dialog zur Einstellung von Darstellungsoptionen

Die Generierung eines Oberflächenmodells lässt sich nun wahlweise über die Schaltfläche ganz rechts im Tool-Bar (vgl. Abbildung 5.2.1) oder über einen Menüpunkt im Menü Generate starten. Alle Berechnungen geschehen bis einschließlich der Anzeige des Modells dann ohne Eingriff des Benutzers. Wie bereits in Kapitel 4 erwähnt, können jedoch vorher Optionen für das Vorgehen festgelegt werden. Abbildung 5.2.3 auf der folgenden Seite zeigt den Inhalt des Menüpunkts Generate*:* Aktuell eingestellte Optionen werden durch einen Punkt davor (genauer gesagt einem *radio button)* markiert, in der Abbildung sind dies die initialen Einstellungen.

Abbildung 5.2.3: Einstellungsmöglichkeiten für die Generierung von Oberflächenmodellen

Der Benutzer kann entscheiden, ob alle *self-loops* automatischer Ereignisse gelöscht werden sollen oder nur diejenigen, welche wirklich überflüssig sind, weil das zugehörige Ereignis ausschließlich in Form solcher *self-loop*-Transitionen vorkommt. Weiter sind im Programm zwei verschiedene Möglichkeiten umgesetzt, die Zustände des Oberflächenmodells automatisch anzuordnen (die Anordnung lässt sich hinterher noch manuell verändern): entweder es wird einzig die Verteilung der zusammengefassten Zustände innerhalb einer Klasse als Kriterium zugrunde gelegt oder die Zustandsanordnung wird zusätzlich noch gerastert. Es sei angemerkt, dass beide Alternativen lediglich Heuristiken darstellen, die von der Annahme ausgehen, dass der Benutzer die Zustände des Maschinenmodells *sinnvoll* ausgerichtet hat. Es hängt vom Einzelfall ab, welche Methode das bessere Ergebnis liefert. Als drittes findet sich noch die Auswahl, ob der originale Reduktionsalgorithmus von Heymann und Degani angewendet werden soll, der – wie in Kapitel 4.4 gezeigt wurde – nicht zwangsweise fehlerfrei ist, oder ob anstatt dessen die geänderte Version auszuführen ist, in der *restricting states* und *concealing states* vermieden werden.
Zuletzt ist noch zu sagen, dass das Software-Tool diverse Fehler verhindert und mittels Dialogen Rückmeldung darüber gibt. So ist es exemplarisch nicht möglich, nicht-deterministische Transitionen oder Zustände gleichen Namens zu erstellen, ebenso wird etwa die korrekte Eingabe von Zahlenwerten überprüft. Darüber hinaus wird bei kritischen Operationen, wie dem Entfernen von Zuständen oder dem Beenden des Programms ohne Speichern, eine Bestätigung gefordert. Weiter wird der Benutzer bei der Bearbeitung des Maschinenmodells angemessen unterstützt, er erhält zum Beispiel bei Änderung des Namens einer Transition die Wahlmöglichkeit, nur genau diese Transition zu editieren oder direkt alle gleichen Namens. Außerdem sind – wie im vorigen Teilkapitel angesprochen – stets nur die Funktionen zugänglich, die aktuell sinnvoll sind. Existiert etwa keine einzige Spezifikationsklasse, so sind die Schaltflächen für das Hinzufügen von Zuständen zu einer Klasse oder das Entfernen einer solchen unnütz und werden entsprechend temporär deaktiviert.

Nachdem der Editor samt seinen Funktionen ausreichend beschrieben worden ist, wird die Benutzung des Programms abschließend in den einzelnen Schritten demonstriert, die zur Erstellung eines Maschinenmodells und der darauf folgenden Generierung und Nachbearbeitung des Oberflächenmodells zu vollführen sind. Das nun anstehende letzte Teilkapitel lässt sich somit als eine Art Kurzanleitung für das implementierte Software-Tool begreifen.

5.3 EXEMPLARISCHE ANWENDUNG DER SOFTWARE

Die Benutzung des Editors wird im Folgenden anhand eines simplen Beispiels gezeigt, das die Erstellung von Zuständen, Transitionen und Spezifikationsklassen sowie die Generierung des Oberflächenmodells und das Speichern beider Modelle erklärt. Der Anschaulichkeit halber werden als bevorzugte Steuerungselemente hier die Schaltflächen im Tool-Bar verwendet.
Beim Start der Software kann der Benutzer prinzipiell nur zwei mögliche Vorgehensweisen verfolgen; entweder lädt er ein bestehendes Modell oder er erstellt den ersten Zustand eines neuen Maschinenmodells. Alle anderen Funktionen (abgesehen von Optionen) machen ohne vorhandenes Modell keinen Sinn und so blendet die Oberfläche sie aus (vgl. Abbildung 5.3.1).

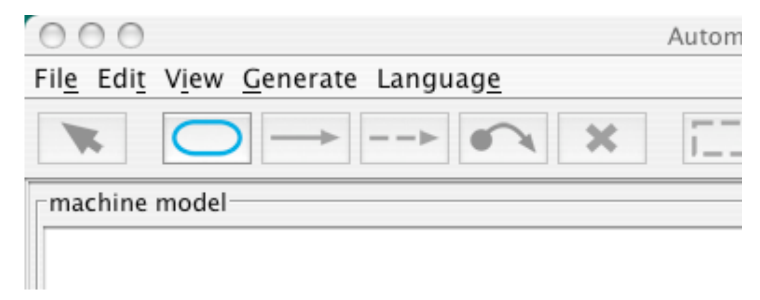


Abbildung 5.3.1: Zu Beginn ist das einzig verfügbare Werkzeug „Neuen Zustand erstellen“

Ein Klick auf die Arbeitsfläche des Maschinenmodells öffnet nun ein Fenster, in dem ein Name für den zu erstellenden Zustand einzugeben ist. Nacheinander seien hier nun die Zustände State1, State2, State3 und State4 erzeugt. Abbildung 5.2.3 stellt die Situation dar, in der schon drei der vier Zustände existieren und ein weiterer gerade hinzugefügt wird. Es zeigt sich, dass bereits hier weitere Werkzeuge anwählbar sind. Außerdem ist erkennbar, dass der erste Zustand automatisch als vorläufiger Startzustand festgesetzt wurde.

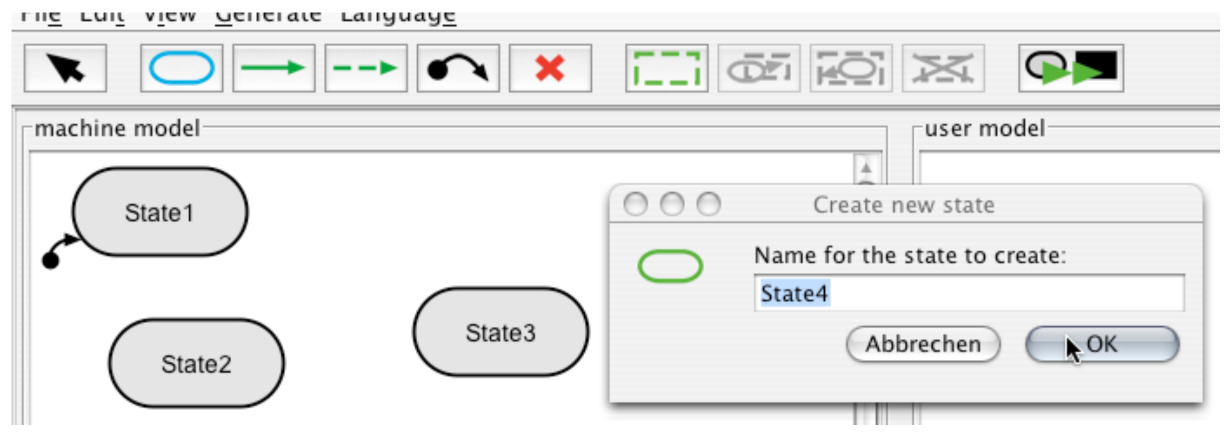


Abbildung 5.3.2: Drei Zustände existieren bereits; ein vierter wird gerade erstellt

Als nächstes sollen die Zustände durch Transitionen miteinander verbunden werden. Die beiden Schaltflächen rechts neben der für neue Zustände (vgl. Abbildung 5.3.2) dienen genau dazu. Vorher seien die Zustände aber noch in eine übersichtlichere Anordnung gebracht. Dazu wird das Standardwerkzeug ausgewählt (Schaltfläche ganz links), die Zustände lassen sich damit per *drag'n'drop* bewegen. Nun werden einige benutzergesteuerte Transitionen erzeugt. Dies geschieht, indem erst der Zustand, von dem die Transition ausgeht, und anschließend der zugehörige

Zielzustand angeklickt wird. Die Vergabe eines Namens erfolgt äquivalent zu der bei Zuständen. Abbildung 5.2.4 zeigt den Moment, wo der Benutzer den ersten Zustand (State4) der letzten zu erstellenden Transition, die wie schon eine andere *t3* heißen soll, angeklickt hat und sich mit der Maus über dem zweiten (State3) befindet. Zur Unterstützung des Gedächtnisses werden in einem solchen Vorgang bereits zugewiesene Zustände grün eingefärbt.
Der Verlauf der Linien der Transitionen wird automatisch festgelegt. Hierbei findet eine Unterscheidung zwischen 18 verschiedenen Fällen der relativen Lage der beiden betroffenen Zustände statt, die regelt, ob eine Transition gerade, um die Ecke oder in Stufenform verläuft. Mithilfe des Standardwerkzeugs lassen sich übrigens auch einzelne Segmente der Linien von Transitionen verschieben. Auch dies ist mittels *drag'n'drop* umzusetzen.

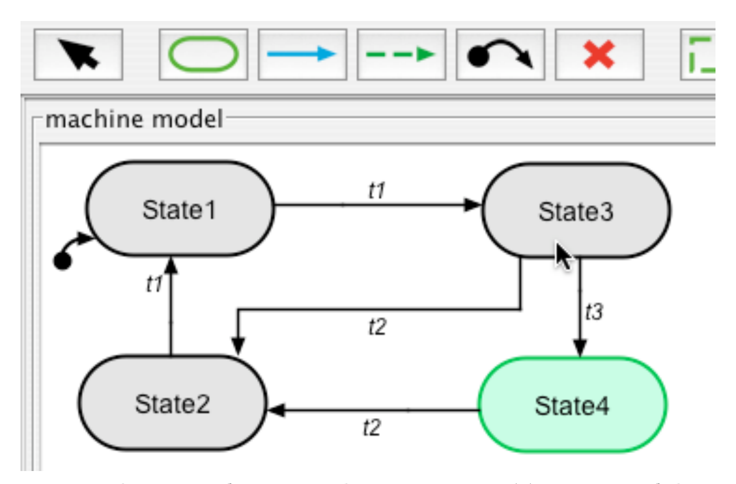


Abbildung 5.3.3: Der Startzustand einer neuen benutzergesteuerten Transition wurde bereits ausgewählt

Es sei nun im Nachhinein entschieden worden, dass *t3* ein automatisches Ereignis bezeichnen soll. Um diese Änderung zu bewerkstelligen, wird eine der zugehörigen Transitionen ausgewählt. Im unteren linken Bereich der Oberfläche erscheinen daraufhin ihre Attribute. In der Auswahlliste type kann dann der Wert automatic eingestellt werden. Betätigen des Buttons change führt die gewünschte Aktion durch. Da es jedoch eine zweite Transition mit demselben Namen gibt, öffnet sich ein Dialog (Abbildung 5.3.4), der hier durch Drücken von Change all bestätigt werde.

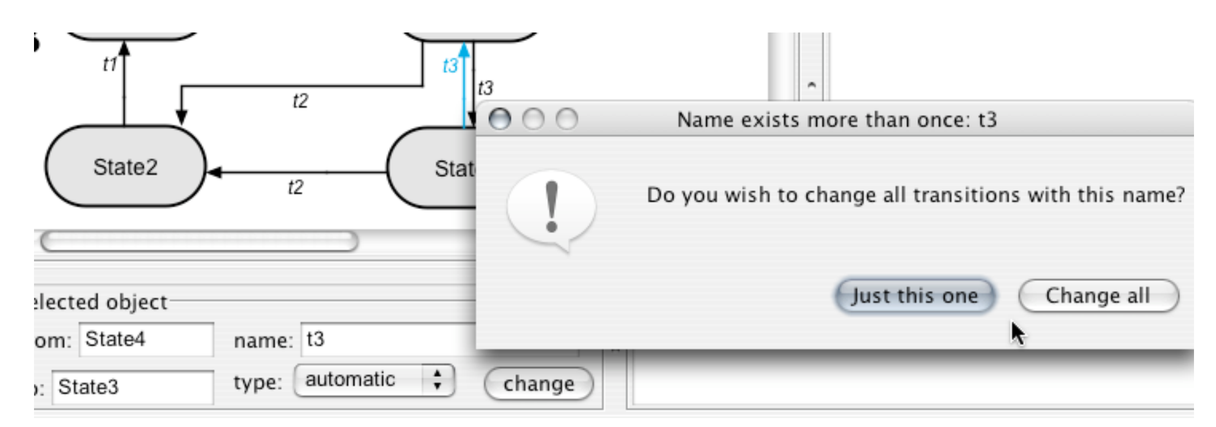


Abbildung 5.3.4: Ändern des Typs von Transitionen

Anschließend wird noch der Startzustand als State2 neu festgelegt, dafür wird das entsprechende Werkzeug im Tool-Bar ausgewählt und der Zustand angeklickt.
Eine Funktion, die am Beispielmodell nicht ausgeführt werden soll, aber exemplarisch einmal gezeigt wird, ist das Löschen eines Zustands oder einer Transition; repräsentiert wird sie durch die Schaltfläche mit dem Kreuz im Tool-Bar. Ein Klick auf das zu entfernende Objekt genügt; es wird jedoch nachgefragt, ob der Löschvorgang wirklich durchgeführt werden soll, dargestellt in Abbildung 5.2.5. Diese Frage werde hier verneint.

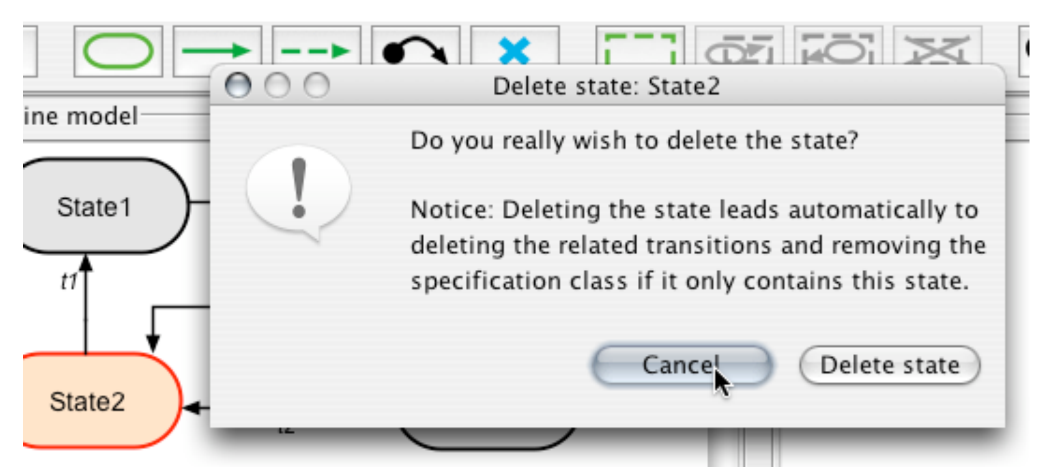


Abbildung 5.2.5: Entfernen eines Objekts, hier des rot eingefärbten Zustands, erfordert eine Bestätigung

Sind alle Zustände und Transitionen vorhanden, so ist der nächste logische Schritt, die Einteilung in Spezifikationsklassen vorzunehmen. Dazu wird die Schaltfläche mit dem gestrichelten grünen Kasten gedrückt. Davon ausgehend werden die zusammenzufassenden Zustände per Linksklick ausgewählt, die Eingabe wird dann per Rechtsklick (unter *Mac: „[ctrl]"* + Linksklick) beendet. Ein Fenster erscheint, in dem der Name der Klasse einzugeben ist. Für das vorliegende Beispiel sei eine Partition der Zustände in {State1, State2} und {State3, State4} vorgenommen. Benannt werden die Klassen hier entsprechend **1 AND 2** und **3 AND 4**. Die Erstellung der ersten Spezifikationsklasse ist in Abbildung 5.2.6 zu sehen.

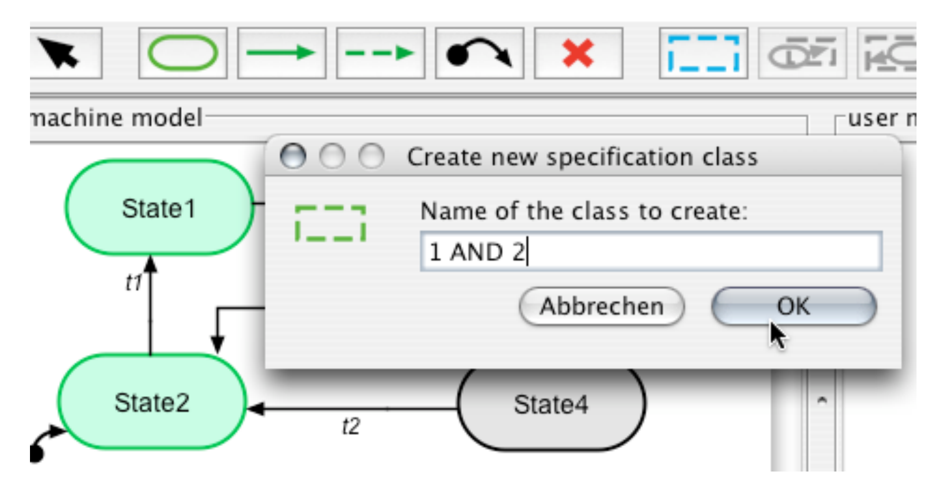


Abbildung 5.2.6: Erstellung einer Spezifikationsklasse

Als Konsequenz der Existenz von Spezifikationsklassen sind alle Schaltflächen des Tool-Bars aktiviert. Neben der Schaltfläche „Neue Klasse erstellen" finden sich von links nach rechts

Schaltflächen für das Hinzufügen eines Zustands zu einer Klasse, das Entfernen eines Zustands aus einer Klasse und das Löschen einer Klasse (Abbildung 5.2.7). Ihre Bedienung wird hier nicht vorgestellt, eine Kurzbeschreibung des dabei nötigen Vorgehens ist aber wie angesprochen im unteren rechten Bereich der Oberfläche nachlesbar.



Abbildung 5.2.7: Vier Schaltflächen (rechts) für den Umgang mit Spezifikationsklassen

Die Konstruktion des Maschinenmodells endet an diesem Punkt. Demzufolge kann jetzt das Oberflächenmodell generiert werden. Hierfür wird unter dem Menüpunkt Generate zuvor noch die Funktion Rasterize arrangement of states eingestellt, danach führt Drücken des Buttons ganz rechts im Tool-Bar zur Berechnung und Anzeige des korrekten und minimalen Oberflächenmodells. Es ist in Abbildung 5.2.8 dargestellt, die auch die automatische Benennung der Zustände illustriert. Offensichtlich haben sich die Zustände aus **3 AND 4** entgegen denen aus **1 AND 2** als kompatibel herausgestellt. Die genaue Zusammensetzung der Zustände des Maschinenmodells in denen des Oberflächenmodells kann in der Auswahlliste contains nachvollzogen werden, welche für einen ausgewählten Zustand des Oberflächenmodells unten links angezeigt wird.

Namen und Positionen der Zustände und Transitionen ließen sich jetzt noch nach gleichem Vorgehen wie beim Maschinenmodell nachträglich ändern, was hier nicht geschehen soll. Es sei aber darauf hingewiesen, dass die Umbenennung einer der mit *t1* gelabelten Transition gleichzeitig die Umbenennung aller so gelabelten Transitionen bewirkt, um die Semantik des Modells aufrecht zu erhalten.

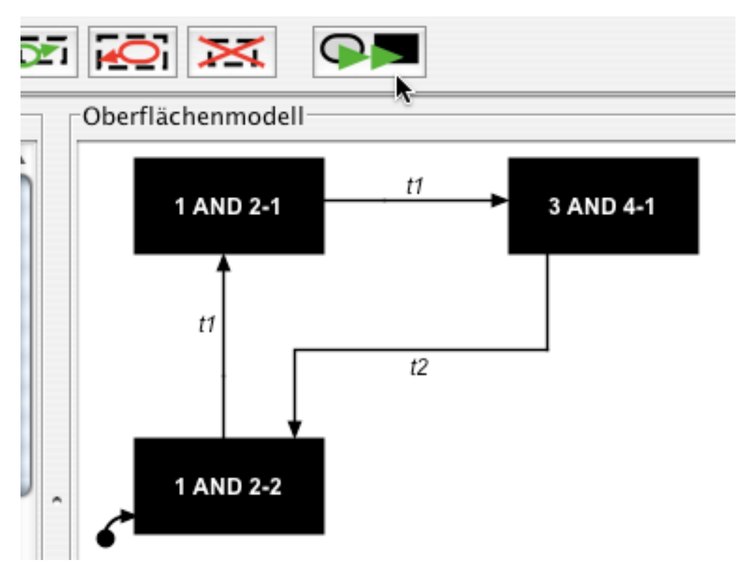


Abbildung 5.2.8: Drücken des Buttons oben in der Mitte führt zur Generierung des Oberflächenmodells

Die letzte hier noch beschriebene Aktion soll das Speichern der beiden Modelle sein. Dazu wird Save model unter dem Menüpunkt File ausgewählt. Es öffnet sich ein Dialog, in dem der gewünschte Ort zum Sichern bestimmt und der Name der Datei eingegeben werden kann. Als Default-Extension wird wie erwähnt **.mod* angezeigt, im Bedarfsfall kann aber auch eine andere Endung benutzt werden. Beim nächsten Lade- oder Speichervorgang wird der zuvor ausgewählte Ordner direkt wieder aufgerufen.

Das vorgestellte Beispiel sollte als Einstieg in die Bedienung des Software-Tools dienen. Es hat einen typischen Ablauf beim Entwurf des Maschinenmodells aufgezeigt, der sich aber genauso gut an vielen Stellen abwandeln ließe. Darüber hinaus sind nicht alle Bedienelemente und Funktionen zum Tragen gekommen; nichtsdestotrotz müsste die Benutzung des Programms von hier an intuitiv und ohne weitere Erklärung möglich sein. Zusammen mit der Java-nahen Beschreibung der internen Strukturen und Abläufe und dem Überblick über die Funktionen seien die Eigenschaften der Software damit ausreichend behandelt.
Das Ende dieses Kapitels bezeichnet im Grunde ebenso das Ende des inhaltlichen Teils der Arbeit. Das folgende Kapitel fasst die wesentlichen Erkenntnisse und Ergebnisse dieser Arbeit noch einmal zusammen. Außerdem wird ein Ausblick auf mögliche Erweiterungen der Software sowie insbesondere auf noch zu behandelnde Themen bezüglich des Verfahrens zur Generierung von korrekten und minimalen Oberflächenmodellen gegeben.

6 ZUSAMMENFASSUNG UND AUSBLICK

„[...] good interface design is all about revealing the underlying structure to the user (and averting surprise)."

(DEGANI, A. 2004, S. 138)

Menschen arbeiten mit Maschinen, um bestimmte Aufgaben zu erfüllen. Fehler treten dort auf, wo der Benutzer nicht versteht, was im System passiert, oder wo die kognitive Belastung die Fähigkeiten des Benutzers übersteigt. Beobachtbarkeit stellt sich als zentrale Qualität für die Vorbeugung gegenüber solchen Gefahrenquellen heraus: Der Benutzer muss auf der einen Seite zu jedem Zeitpunkt ausreichend darüber bescheid wissen, in welchem Zustand sich das unterliegende System befindet und welchen es in Abhängigkeit zur nächsten Aktion oder möglichen internen Ereignissen erreicht. Ein System, das diesen Forderungen gerecht wird, heißt korrekt. Auf der anderen Seite sollten für die Bedienung irrelevante Informationen ausgeblendet werden, da sie dem Zweck der Konzentration auf die Aufgabe entgegenstehen (Kapitel 2).
Technische Geräte und Programme wie auch deren User Interfaces lassen sich für die Analyse der Beobachtbarkeit als Zustandsübergangsdiagramme modellieren. Weiter stellt die Einteilung der Zustände des Systems in Spezifikationsklassen eine Formalisierung der Aufgabenspezifikation dar, die besagt, welche Zustände für einen Benutzer zumindest inhaltlich das gleiche bedeuten. Gegeben ein zustandsbasiertes Modell eines Systems samt einer solchen Aufgabenspezifikation, bietet eine von Michael Heymann und Asaf Degani vorgestellte Methode die Möglichkeit, das Modell einer Oberfläche auf seine Korrektheit hin zu verifizieren (Kapitel 3).

Bislang basiert der übliche Prozess beim User Interface Design mehr oder minder auf intuitivem Vorgehen in Kombination mit empirischen Heuristiken. Die stetig ansteigende Bedeutung und Komplexität von technischen Systemen in Alltag, Forschung und sicherheitskritischen Bereichen verdeutlicht aber, dass ein Wandel zur automatisierten Generierung von Interfaces in nicht ferner Zukunft erforderlich sein wird. Heymann und Degani haben ein Konzept entwickelt, aus dem Modell der Maschine das Modell der Oberfläche formal abzuleiten. Das entsprechende Verfahren wurde sowohl abstrakt als auch anschaulich beschrieben. Ihm liegen mathematische Algorithmen zugrunde, die spezifikationsäquivalente Zustände (auch „kompatibel" genannt) berechnen und zusammenfassen und aus diesen eine Überdeckung des Maschinenmodells bilden (Kapitel 4.1 bis 4.3).
Es wurde jedoch gezeigt, dass das Konzept Schwachstellen birgt, also die Möglichkeit besteht, dass das resultierende Interface zu nicht genügend beobachtbaren Situationen führen kann. Für dieses Problem wurde anschließend Abhilfe geleistet, indem einerseits eine zusätzliche Bedingung für Spezifikationsklassen gefordert wurde, andererseits aber auch eine wirkliche Änderung des

Reduktionsalgorithmus durchgeführt worden ist. Aus den Fehlerbehebungen folgt die bewiesene Korrektheit des Verfahrens (Kapitel 4.4 und 4.5).
Dennoch bleiben noch weiter zu behandelnde Aspekte offen: So wurde für die Vermeidung von Zuständen, in denen der Benutzer in Unkenntnis über eine für ihn mögliche Aktion ist *(restricting states),* nur ein provisorischer Lösungsansatz geliefert. Verbesserungen des Reduktionsalgorithmus diesbezüglich werden mit der Überarbeitung seiner Kernberechnungen einhergehen.
Darüber hinaus beinhaltet das Verfahren Einschränkungen, die noch nicht konkreter untersucht worden sind, deren Entfernung aber der Anwendbarkeit zuträglich wäre. Zum Beispiel ist die Frage nicht geklärt, wie mit zeitlich gesteuerten Ereignissen *(timed events)* umzugehen ist. Wie kann der Algorithmus etwa erkennen, dass es einen Zusammenhang zwischen „after three seconds" und „after five seconds" gibt und wie muss auf dies reagiert werden? Und mehr noch: Wie kann abgesichert werden, dass zeitliche Kopplungen für einen Benutzer beobachtbar sind? Unabhängig solcher Schwierigkeiten bestände – wie schon am Ende von Kapitel 4 implizit angesprochen – großer Nutzen in der Portierung des Verfahrens auf andere Modelltypen, etwa auf Petri-Netze.
Sicherlich lassen sich noch einige andere Forschungsansätze für die vorliegende Thematik finden, aber bereits die angesprochenen Punkte legen offen, dass noch Handlungsbedarf besteht.

Neben der schriftlichen Bearbeitung der angesprochenen Inhalte wurde im Rahmen dieser Arbeit ein Software-Tool implementiert, das den Reduktionsalgorithmus umsetzt. Das Tool ermöglicht die manuelle Erstellung von Maschinenmodellen sowie die Einteilung in Spezifikationsklassen komfortabel auf graphischer Ebene. Die Generierung des zugehörigen Oberflächenmodells verläuft automatisch, sowohl was die Berechnung als auch was die Anordnung der Zustände betrifft. Außerdem erlaubt das Programm, die in bestimmten Fällen auftretenden Fehler des originalen Algorithmus zu vermeiden (Kapitel 5).
Mehrere Erweiterungen des Software-Tools sind denkbar. So könnte etwa das beschriebene Verifikationsverfahren noch eingebettet werden. Die grundlegenden Voraussetzungen dafür sind bereits geschaffen, denn nur geringe Änderungen am Quellcode würden das ebenfalls manuelle Erzeugen von Oberflächenmodellen gestatten. Darauf aufbauend könnte ein eigenes Modul für die Konstruktion des zusammengesetzten Modells implementiert werden.
Ferner könnte eine Speicherung der Modelle im *XML*-Format nützlich sein, um eine bessere Interoperabilität mit anderen Modellierungswerkzeugen zu erreichen. Auch hierfür bedarf es keiner wesentlichen Modifikationen des bestehenden Codes, findet doch die Behandlung programmexterner Daten in einer einzigen Klasse statt.
Schließlich lässt sich die *Usability* der bestehenden Implementierung zweifelsohne noch steigern. Zwar war ein Ziel beim Entwurf der Software, ein intuitiv und leicht bedienbares User Interface anzubieten, und dieses Vorhaben wurde auch erfolgreich vollzogen. Dennoch fehlen einzelne der Benutzbarkeit förderlichen Produktcharakteristiken, wie etwa eine *Undo-*/*Redo*-Funktion oder eine umfassendere Online-Hilfe. Auch würde eine Erweiterung der Darstellung der Modelle um die Notation als Statechart-Diagramme dem Benutzer helfen, komplexe Systeme besser überschauen zu können. Derlei Qualitätsmerkmale wurden bisher nicht umgesetzt, weil sie nicht der direkten Thematik angehören und den Umfang der Arbeit überstiegen hätten.

ANHANG

EIDESSTATTLICHE ERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe sowie ohne Verwendung anderer als der im Literaturverzeichnis angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Inhalt der Arbeit hat meines Wissens in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

____

Paderborn, den 10. November 2006

AUF DER BEIGEFÜGTEN CD ENTHALTEN

Dieser Arbeit ist eine CD-ROM beigefügt, auf der sich das erstellte Software-Tool und weitere Materialien befinden. Im Einzelnen enthält die CD die folgenden Daten:

- Auf oberster Ebene liegt die implementierte Software als ausführbare *JAR*-Datei mit dem Namen ModelTool_1_0.jar sowie eine *PDF*-Version der vorliegenden Arbeit, die WachsmuthSA.pdf heißt.
- Im Ordner examples sind die in der Arbeit verwendeten Beispielmodelle zur Verfügung gestellt, benannt mit Teilkapitelnummer und inhaltlicher Thematik, so etwa 4_4_augmenting_state. Ebenso befinden sich dort die drei Beispiele aus HEYMANN & DEGANI 2002 I und HEYMANN & DEGANI 2006.
- Der Ordner javadoc beinhaltet die aus dem Quellcode generierten *Javadoc*-HTML-Dateien mit der Startseite index.html.
- In references finden sich Kopien aller verwendeten Quellen, die im Internet zum Download bereit stehen.
- Der Ordner source enthält schließlich den Java-Quellcode des Programms, der entsprechend der Packages in Unterverzeichnisse aufgeteilt ist.

LITERATURVERZEICHNIS

DEGANI, ASAF (2004): Taming HAL: Designing Interfaces Beyond 2001. Palgrave Macmillan, New York.

DEGANI, ASAF & HEYMANN, MICHAEL (2002): Formal Verification of Human-Automation Interaction. Human Factors, 44(1), S. 28-43.
http://ti.arc.nasa.gov/people/asaf/hai/pdf/Formal%20Verification%20of%20Human-Automation%20Interaction.pdf, 09.10.2006.

HAREL, DAVID (1987): Statecharts – A Visual Formalism for Complex Systems. Science of Computer Programming 8, S. 231 –274.

HEYMANN, MICHAEL & DEGANI, ASAF (2002 I): On Abstractions and Simplifications in the Design of Human-Automation Interfaces. NASA Technical Memorandum #211397. Moffett Field, CA: NASA Ames Research Center.
http://ti.arc.nasa.gov/people/asaf/interface_design/pdf/NASA%20Report%20--%20On%20Abstractions%20and%20Simplifications%20in%20Design.pdf, 29.09.2006.

HEYMANN, MICHAEL & DEGANI, ASAF (2002 II): Constructing Human-Automation Interfaces: A Formal Approach. Ninth International Conference on Human-Computer Interaction in Aeronautics. Boston: Massachusetts Institute of Technology.
http://ti.arc.nasa.gov/people/asaf/interface_design/pdf/Constructing%20Human-Automation%20Interfaces.pdf, 29.09.2006.

HEYMANN, MICHAEL & DEGANI, ASAF (2002 III): On the Construction of Human-Automation Interfaces by Formal Abstraction. In S. Koenig and R. Holte (Eds.), Abstraction, Reformulation and Approximation, S. 99-115. London, UK: Springer-Verlag.
http://ti.arc.nasa.gov/people/asaf/interface_design/pdf/On%20the%20Construction%20of%20Human-Automation%20Interfaces.pdf, 29.09.2006.

HEYMANN, MICHAEL & DEGANI, ASAF (2006): Formal Analysis and Automatic Generation of User Interfaces: Approach, Methodology, and an Algorithm. Paper accepted for publication, Human Factors Journal. Expected date of publication, Spring 2007.
http://ti.arc.nasa.gov/people/asaf/interface_design/pdf/Generation_of_User_Interfaces.pdf, preprint, 29.09.2006.

NORMAN, DONALD A. (1989): The Design of Everyday Things. Paperback edition. Currency/Doubleday, New York.

PARASURAMAN, R., SHERIDAN, T.B. & WICKENS, C.D. (2000): A Model for Types and Levels of Human Interaction with Automation. IEEE Transaction on Systems, Man, and Cybernetics – Part A: Systems and Humans, 30(3).
http://www.cs.uml.edu/~holly/91.549/readings/sheridan-autonomy.pdf, 29.09.2006.